A Compiler Infrastructure for Research on High-Performance Java

by

Neil V. Brewster

A Thesis submitted in conformity with the requirements
for the Degree of Master of Applied Science,
Graduate Department of Electrical and Computer Engineering,
in the University of Toronto

A Compiler Infrastructure for Research on High-Performance Java

Neil V. Brewster
Master of Applied Science, 2001
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

This thesis describes the zJava High Level Intermediate Representation (HLIR), which provides a framework for the analysis and restructuring of Java programs at the source code level. The system is designed to minimize the time taken to prototype new compiler analyses, guaranteeing under transformations both the consistency of its internal structure and the syntactic correctness of the represented code. We address several challenges unique to Java, which have not been addressed by earlier frameworks. These include automatic maintenance of complex symbol scope information under transformations, insertion of implicit code to accurately model the source program, incorporation of compiled code into the representation, and representation of the complex control flow of exception handling constructs. We include support for the sharing of information between compiler passes, and mechanisms to support interprocedural analysis. We believe that the features we introduce in the zJava HLIR will result in a means of rapidly prototyping new Java compiler analyses. We give a number of examples illustrating the use and utility of the infrastructure.

Acknowledgments

My thanks go first to my supervisor, Tarek Abdelrahman, for his guidance, support, and inestimable patience. Thanks also go to Dion Lew and Bryan Chan, other members of the zJava project team.

This work would never have progressed as far as it has without the input of Kit Nguyen, Jonathan Winter, Vivian Quan and David Chung; many thanks to these these students for working with zJava, exposing problems, making many helpful suggestions, and providing concrete examples of our system in use.

Thanks to all my Skule^{TM} friends, for believing me every time I said "one more month!"; especially to Kevin Harris, for keeping me occupied throwing things around the lab late at night, and to Courtney Gibson, for working hard enough on his research to motivate us all.

Finally, I would like to thank my parents and brothers, and Pam, for all their support, without which I would never have made it this far.

Contents

1 Introduction **1**
1.1 Organization 4

2 HLIR Architecture **6**
2.1 Goals 6
2.2 Architecture 8
2.2.1 Flat Representation 8
2.2.2 Robust System 9
2.2.2.1 Syntactic Consistency 9
2.2.2.2 Internal Consistency 11
2.2.3 Modularity of Analyses 12
2.2.3.1 Serialization 12
2.2.3.2 Compiler Directives 13
2.2.3.3 Rich Class Format 13
2.2.4 Control Flow Representation 14
2.2.5 Symbol Tables 15
2.2.6 Implicit Execution 17
2.2.6.1 Variable Initializers 17
2.2.6.2 Implicit Constructor Invocation 18
2.2.6.3 Instance Variables and Instance Initializer Blocks . . 18
2.2.6.4 Class Variables and Class Initializer Blocks 19
2.2.6.5 Implicit Constructor Declaration 21
2.2.7 External Class Loading 21
2.2.8 Inner Classes 22
2.2.9 Source Code Generation 23
2.2.10 Attributes 23
2.3 Related Work 23
2.3.1 High-Level Frameworks 23
2.3.2 Bytecode IRs 26

3 HLIR Classes **30**
3.1 Program 30
3.2 Compilation Unit 30
3.2.1 External Class Resolution 31
3.2.1.1 Search Sequence 32

3.2.1.2 Class Accessibility . . . 32
3.3 Classes . . . 33
3.4 Methods . . . 34
3.5 Statements . . . 35
3.5.1 Single Statements . . . 37
3.5.1.1 Expression Statement . . . 38
3.5.1.2 Break and Continue Statements . . . 38
3.5.1.3 Implicit Statements . . . 38
3.5.1.4 Declaration Statements . . . 38
3.5.2 Compound Statements . . . 39
3.5.2.1 Block Header . . . 39
3.5.2.2 If Construct Header . . . 39
3.5.2.3 Labelled Construct Header . . . 39
3.5.2.4 Switch Construct Header . . . 40
3.5.2.5 Try-Catch-Finally Header . . . 40
3.5.2.6 Synchronized Block Header . . . 40
3.5.3 Iteration Statements . . . 40
3.5.3.1 While Loop Header . . . 41
3.5.3.2 Do Loop Header . . . 41
3.5.3.3 For Loop Header . . . 41
3.5.4 Statement List . . . 41
3.5.4.1 Block Construct . . . 43
3.5.4.2 Labelled Construct . . . 44
3.5.4.3 Try-Catch-Finally Construct . . . 44
3.5.4.4 Switch Construct . . . 46
3.5.4.5 If-Then-Else Construct . . . 47
3.5.4.6 Synchronized Block . . . 47
3.5.4.7 Do and While Loops . . . 48
3.5.4.8 For Loop . . . 48
3.5.5 Statement List Consistency . . . 50
3.5.5.1 Statement List Insertion . . . 50
3.5.5.2 Statement List Removal . . . 52
3.6 Expressions . . . 54
3.6.1 Expression classes . . . 54
3.6.1.1 Array Access Expression . . . 55
3.6.1.2 Binary Expression . . . 56
3.6.1.3 Type Comparison Expression . . . 56
3.6.1.4 Cast Expression . . . 57
3.6.1.5 Field Access Expression . . . 57
3.6.1.6 Variable Expression . . . 58
3.6.1.7 Method Invocation Expression . . . 58
3.6.1.8 Array Creation Expression . . . 58
3.6.1.9 Class Instance Creation Expression . . . 59
3.6.1.10 Unary Expressions . . . 59
3.6.1.11 `super` and `this` Expressions . . . 59

3.6.1.12 The Conditional ? : operator . . . 59
3.6.2 Type Evaluation . . . 60
3.6.2.1 Qualified Expressions . . . 61
3.6.2.2 arrayAccess . . . 62
3.6.2.3 binaryExpression . . . 62
3.6.2.4 instanceOfExpression . . . 63
3.6.2.5 castExpression . . . 63
3.6.2.6 fieldAccess . . . 63
3.6.2.7 varExpression . . . 66
3.6.2.8 methodInvocation . . . 67
3.6.2.9 newArray . . . 68
3.6.2.10 newObject . . . 68
3.6.2.11 preUnaryExpression and postUnaryExpression . . . 68
3.6.2.12 superExpression and thisExpression . . . 68
3.6.2.13 questionExpression . . . 68
3.6.2.14 Type Conversion and Promotion . . . 69
3.6.2.15 Nested Classes . . . 70
3.6.3 Arrays . . . 70
3.6.4 Source Code Regeneration . . . 71
3.7 Symbols . . . 72
3.8 Symbol Table . . . 73
3.8.1 Incremental Symbol Table Updates . . . 76
3.8.2 Superclass Symbols . . . 79
3.9 Access Modifiers . . . 79
3.10 Types . . . 80

4 Support Classes **82**
4.1 The zJava Frontend . . . 82
4.2 zJavaUser . . . 83
4.3 zObject . . . 83
4.3.1 zAttribute . . . 84
4.3.2 Source-to-Source Transformation . . . 84
4.4 zLinkedList and zRefList . . . 84
4.5 zHashTable . . . 85
4.6 zClassLoader . . . 86
4.7 zJavaException . . . 87
4.8 zDirective . . . 87

5 Examples **90**
5.1 Statement Insertion . . . 90
5.2 Control Flow Graph . . . 91
5.2.1 break/continue . . . 93
5.2.2 return . . . 94
5.2.3 if-then-else . . . 94
5.2.4 switch . . . 95

5.2.5 label . . . 95
5.2.6 Iteration Constructs . . . 95
5.2.7 Exceptions . . . 97
5.2.7.1 Java Exception Model . . . 97
5.2.7.2 Try-Catch-Finally . . . 98
5.2.7.3 Propagating Jumps . . . 100
5.2.7.4 Potential Exception-Causing Instructions . . . 101
5.2.7.5 Precise Modelling of Exceptions . . . 102
5.3 Data Structure Visualizer . . . 102
5.4 Software Architecture Visualization . . . 104

6 Conclusions **107**
6.1 Future Work . . . 108
6.1.1 HLIR-BCIR links . . . 108
6.1.2 Semantic Consistency . . . 108
6.1.3 Control Flow . . . 108
6.1.4 Java1.1 . . . 109
6.1.5 Builder Methods . . . 109
6.1.6 Support Classes . . . 109
6.1.7 Type Conversions . . . 110

A Testing and Verification **111**
A.1 Self-Compile Test . . . 111
A.2 SPEC Benchmarks . . . 112
A.3 Lines of Code . . . 113

B The zDebug Interface **114**

C zJava HLIR API **117**

Bibliography **117**

List of Figures

1.1 Overview of the zJava compiler infrastructure. 2

2.1 Symbol table tree example, showing inheritance and hiding. 16
2.2 Examples of implicit code added in to a class. Implicitly added code is emphasized. 20
2.3 The implicit initialization methods generated for class Foo from Figure 2.2. 21

3.1 A simple compilation unit. 33
3.2 Class hierarchy of the zStatement package. 36
3.3 Structure of a block construct. 43
3.4 Structure of a labelled construct. 44
3.5 Structure of a try-catch-finally construct with one catch clause. . . . 45
3.6 Structure of a switch construct. 46
3.7 Structure of an if construct. 47
3.8 Structure of a synchronized construct. 48
3.9 Structure of a do loop. 48
3.10 Structure of a for loop. 49
3.11 Examples of two fully-formed statement lists, with equivalent Java source code. 51
3.12 Example of Name disambiguation. 64
3.13 Examples of array declarations, array initializers, and field accesses on an array. 71
3.14 A `for` loop, and the equivalent "flat" representation. 75
3.15 Scoping/symbol table representation for an `if` construct. 75
3.16 Automatic symbol table updates on statement insertion. 78
3.17 Symbol table example, demonstrating the use of **SuperSymbolTable** to represent inheritance. 80

4.1 Examples of compiler directives. 88

5.1 Using HLIR to construct a `for` loop. 92
5.2 Control flow representation of an `if-then-else` construct. 94
5.3 Control flow in a *switch* construct. 96
5.4 Control flow in an *iteration* construct. 97
5.5 Control flow in a *try* construct. 99
5.6 Using HLIR to add DSV library calls each time a node is allocated. . 104

List of Tables

2.1 Comparison of various high-level compiler frameworks. A filled circle represents that the system has the associated functionality. 27

3.1 Single statements (not part of a construct), with references to JLS [4] sections. 37
3.2 Allowable construct removals 53
3.3 Sub-constructs, which can be removed via special methods in the associated construct header 53
3.4 Expressions which contain no subexpressions. 55
3.5 Expressions which may contain subexpressions. 56
3.6 Various binary expressions represented by **binaryExpression**. 57
3.7 Expressions which contain no subexpressions, with types. 61
3.8 Type representation in HLIR 81

5.1 A subset of the low-level relation types for Java [52]. 105

B.1 Additional commands available at the *compilation unit* level. 115
B.2 Additional commands available at the *class* level. 115
B.3 Additional commands available at the *method* level. 115
B.4 Additional commands available at the *statement* level. 116
B.5 Additional commands available at the *expression* level. 116

C.1 Several methods of the **compilationUnit** class 117
C.2 Several methods of the **classObject** class, including some inherited from **ClassShell**. 118
C.3 Several methods of the **methodObject** class, including some inherited from **MethodShell**. 118
C.4 Several methods of the **zStatement** class 119
C.5 Several methods of the **stmtList** class 119
C.6 Several methods of the **zExpression** class 120
C.7 Several methods of the **CFGNode** class 120

Chapter 1

Introduction

There has been considerable research in the past decade on parallelizing compilers and automatic parallelization of programs [1][2][3]. Traditionally, research in automatic parallelization focused on scientific applications that consist mainly of loops and array references, typically written in imperative languages like FORTAN or C. With the increasing the popularity of multiprocessor desktop workstations, there is growing interest in the automatic parallelization of general-purpose[1] programs, including those written in object-oriented languages such C++ and more recently Java. A property of such programs is the use of pointer-based dynamic data structures, such as linked lists and trees, and recursion—aspects that are not generally handled by existing parallelizing compiler technology.

The Java programming language [4] has many features that lead to easier, less error-prone application development, such as object-orientation, platform independence, strong typing, exception handling and automatic storage management. Java is gaining popularity in the development of both small, web-based applications, and larger application systems.

The goal of the zJava compiler project is to investigate method-level automatic parallelization of general-purpose programs written in the Java programming lan-

[1]It is surprisingly difficult to come up with a generic name to describe programs that utilize dynamic data structures and recursion. The name "non-scientific programs" would imply that the computations carried out by such applications are not for scientific purposes, which is certainly not the case. The term "general-purpose applications" would imply that scientific applications are special purpose, which is also not the case. We elect to use the latter name.

guage. The zJava compiler infrastructure is being developed at the University of Toronto as a program restructuring system for Java. This infrastructure provides both a high-level and low-level representation, and the mechanisms to convert between Java source, the intermediate forms, and the `.class` file format.

An overview of the zJava compiler architecture is given in Figure 1.1. Java source code is parsed by the frontend and converted into the zJava High Level Intermediate Representation[1] (HLIR), on which user analysis and transformation passes are performed. Java source code can be generated from HLIR, allowing analyses to be performed independently and at different times. Compiled code (`.class` files) is loaded by HLIR when needed (e.g. to resolve the datatype of a symbol defined in a library class). HLIR can be converted into the zJava Bytecode Intermediate Representation[1] (BCIR), upon which further analysis passes may occur before Java bytecode is generated. A separate Java compiler (such as Sun's `javac`) can be used to generate `.class` files for input to BCIR, or to compile source code generated by HLIR.

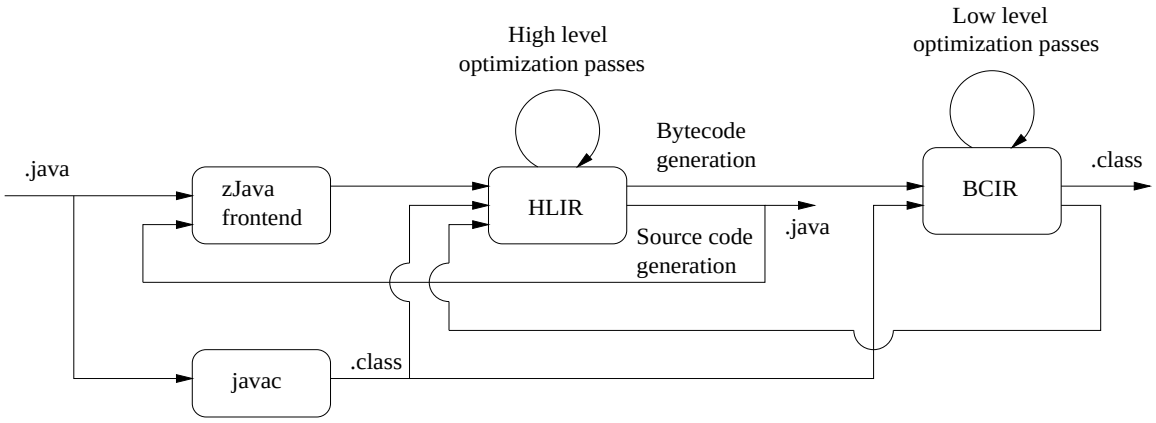


Figure 1.1: Overview of the zJava compiler infrastructure.

This thesis describes the zJava HLIR, which provides a *robust* framework for the *rapid* prototyping of new compiler optimizations and analyses. A high-level representation has the advantage of retaining the syntactic structure of high-level constructs

[1]Note that we use the term "intermediate representation" to mean both the data structures used to represent the code, and the functionality provided to populate and make use of the IR.

(such as loops and switches); this benefits high-level analyses like dependence analysis, array structuring, loop parallelization and transformation, and program restructuring.

There are a number of interesting features of the zJava HLIR:

- *Robustness*. Extensive error checking and reporting speeds up the process of prototyping new compiler passes. The representation is guaranteed to be consistent under transformations. There are two types of consistency that are enforced by the representation:
  - *Syntactic Consistency*. The syntactic correctness of the represented Java program is guaranteed under transformations.
  - *Internal Consistency*. The correctness of the structure of HLIR is enforced, and internal HLIR structures are automatically updated under transformations.
- *Flat representation*. The core of the representation is a flat list of sequential statements for each method body. This facilitates traversal and analysis of the representation, including the grouping of statements into basic blocks for control flow and further analyses.
- *Modularity*. HLIR provides mechanisms to save the results of an analysis for use by later compiler passes. This facilitates modular analysis and the implementation of compiler passes.
- *Class loading*. When source code is not available, classes can be loaded from `.class` files and integrated seamlessly into HLIR.
- *Symbol information*. High-level symbol information, including a precise representation of scoping, is included.
- *Implicit code*. Implicit code, which is not in the source code but is required by the Java Language Specification [4], is represented.

- *Source code generation.* The representation can be converted into human-readable Java source code.

In order to provide these features in the zJava infrastructure, it has been necessary to address a number of challenges not faced by previous high-level representations:

- The zJava HLIR is the first high-level restructuring system for Java, and the first representation of an object-oriented language to include automatic consistency enforcement. Syntactic consistency enforcement, demonstrated by Polaris [5] for Fortran programs, is significantly more complex in a Java representation.

- The representation of symbol scope is made more difficult in Java by inheritance relationships and by the possibility of variable declarations anywhere in a block. We not only represent the complex scope information, but also maintain its correctness under transformations and automatically merge new information when statements are added.

- The automatic insertion of implicit code when converting user code into HLIR is a unique challenge. Without this implicit code, analyses performed on HLIR would be either incomplete, or overly complicated by the inclusion of special cases.

- HLIR includes a framework for passing source code directives into the compiler, and augments this functionality by providing the ability to save analysis results in either source code or bytecode. We also introduce the ability to incorporate compiled code into the high-level representation.

1.1 Organization

The remainder of this thesis is organised as follows: Chapter 2 details the goals of the zJava HLIR, describes the architecture of the representation, and discusses related work. Chapter 3 details classes that make up the core of the IR and are used to represent actual Java constructs such as classes, methods, statements, expressions

and symbols. Chapter 4 details the various support classes that were implemented to realize certain goals. In Chapter 5 we describe some examples that demonstrate the functionality of the zJava HLIR. Chapter 6 presents conclusions and suggestions for future work.

Chapter 2

HLIR Architecture

The purpose of this chapter is to present the principal goals which guided the development of HLIR, and to outline the architectural details which realize these goals. The chapter concludes with a discussion of related work, comparing HLIR architecture features to those of several similar systems.

2.1 Goals

The driving goal of the zJava HLIR is to provide a robust framework for the analysis of Java code at the source level. The most basic goal is the ability to represent source code in the form of an intermediate representation (IR), and to generate source code from the IR. This involves representing all information available in a `.java` file, including symbol names, types, and scope information.

The statement "intermediate representation" is taken to mean more than just the data structures used internally. HLIR also incorporates the functionality required to make use of the representation - the ability to examine, add to, remove from, rearrange, and modify the information. We aim to provide an API for manipulating the IR and for easily developing additional functionality.

A primary goal of the zJava HLIR is to provide a robust system, designed to detect incorrect uses of the IR, and thus minimize the time required to prototype new compiler passes. We aim to detect and report errors as early in the development

of a new pass as is possible, through the use of of compile-time[1] checks. Where that is not possible, Java exceptions are used to report the errors at run-time, with error messages designed facilitate debugging. We define an error as any transformation that would leave the IR in an inconsistent state, both internally and in terms of the syntactic correctness of the represented Java code.

Users of the zJava HLIR are not expected to understand and manipulate all aspects of the representation. We desire that HLIR be designed to properly maintain its internal structures under any transformation, and to disallow any modifications (such as creating, adding, or removing statements or other objects) which would result in an inconsistent representation. Additionally, we provide convenient hooks for users to maintain the consistency of their own structures, without requiring that they modify HLIR itself.

It is often desirable to have a method of saving the results of a lengthy compiler analysis in such a manner that the results of the analysis can be incorporated into a subsequent compiler pass. It is important not to equate this concept of modular analysis to the ability to run several independent compiler passes prior to code generation; the latter is a feature of most program restructuring systems, including the zJava HLIR. Modularity of analyses allows one pass to build on the results of another, *without* having to repeat it. It is a goal of the zJava HLIR to provide the functionality for such modular analyses, including the ability to augment generated `.class` files with analysis results. In this manner we also provide a mechanism for performing interprocedural analyses, which often require information about classes whose source code is not available at compile-time.

HLIR is intended to also represent the code which is not explicit in the source code but which must, according to the Java Language Specification [4], be implicitly added by a compiler. To this end, we include as part of the representation any code which would be implicitly added by a standard Java compiler.

[1]Throughout this thesis, we use the term "compile-time" to refer to the time at which a zJava compiler pass is compiled, and the term "run-time" to refer to the time at which the zJava compiler (and thus the compiler pass) is executed.

Control flow information is considered to be an essential part of the intermediate representation. The zJava infrastructure includes a control flow representation, implemented on top of HLIR and using a design that minimizes the impact of exception flow on the granularity of basic blocks and the number of flow edges, without sacrificing precision.

2.2 Architecture

In this section we describe the architecture design details which achieve the goals introduced above.

2.2.1 Flat Representation

We have chosen to represent each method body as a flat list of non-recursive statement structures. This means that compiler passes can simply iterate over the statements of a method body, rather than traversing a more complicated representation in which certain statements contain other statements. A sequential list of statements is also convenient when breaking a method body up into basic blocks for control flow representation.

In a tree-structured representation, such as the abstract syntax tree (AST) generated by the zJava frontend, the elements of a language construct are usually contained as a subtree of the header statement of the construct. For example, an `if` statement would have the `then` statement and the `else` statement nested within it. In a flat representation, all statements are in a single list. Flattening an AST creates a need for marker statements to represent (at least) the beginning and end of each construct. Thus, in HLIR, each construct begins with a *header* statement and ends with a generic *end* statement. The header and end are linked to each other, and the statements of the construct body appear between the two in the statement list. Each construct header also includes a link to the first statement of its body. Additional links are included, specific to the particular construct; for example, an *if-header* contains a link to the *then-part* and *else-part* in addition to the *if-end* (see Figure 3.7 on page 47). No

information is lost in the transition to a flat list—it is straightforward to determine the nesting structure of the original AST statements.

2.2.2 Robust System

A robust system can significantly reduce the time taken to prototype and debug new compiler analyses. The zJava HLIR is designed to detect and report programmer errors which might leave the IR in an inconsistent state. Wherever possible, the system is designed to catch such errors at compile time. To maintain efficiency and limit code complexity, some checks are performed at run-time. When run-time error catching is the only option, Java exceptions are used to simplify the task of locating and understanding the error. When an error is detected, an exception is thrown, containing a text description of the problem. The error messages associated with each exception, combined with stack traces generated when an error is raised, let the user know what was done in error, and facilitate the process of locating the error in their code.

We have designed HLIR with the goal that it can never be left in an inconsistent state under any transformation. We define consistency both in terms of the state of the internal representation, and in terms of the syntactic correctness of the represented Java source code. We will begin by discussing the latter.

2.2.2.1 Syntactic Consistency

In creating a robust system, our first goal is that of syntactic consistency. We require that HLIR always represent syntactically correct Java code, and that no transformation be permitted which would result in the representation of a syntactically incorrect program.

The syntactic consistency of the representation is primarily enforced by controlling the ways in which lists of statements are created and modified. The set of constructors for creating a statement list require that all parts of a construct be specified at once. For example, it is not possible to create a *while* loop with no body, or an *else* clause with no *if* statement. Additionally, the constructors for each statement type

require that all necessary components be present. It is not possible, for example, to create a *catch clause* header statement without specifying the exception variable. The statement list implementation also strictly controls the way in which statements are added or removed. It is not possible to incrementally build a program by adding one statement at a time, even if the final result is syntactically correct. Instead, the representation can only be built by adding statement lists into other statement lists.

Specific cases which cannot be controlled as described above are handled at run-time in the methods provided for list insertion. The example below demonstrates such a case, showing the insertion of a valid statement list representing a *block* (in this case, an opening cursive brace, a single expression statement, and a closing brace) after a *try block*. The resulting code would be syntactically incorrect if the insertion were to be permitted.

```
try {
  System.out.println(a.foo);
}
{        //
  bar(); //  A valid statement list, in an illegal place!
}        //
catch {NullPointerException e} {
  System.out.println("NPE: " + e);
}
```

Run-time checks also determine when it is permitted to remove statements from a statement list. Removals which would leave a construct in a syntactically invalid state (such as removing the *then-part* of an `if` construct) are not permitted. Thus, only certain single statements, or entire constructs, may be removed. Removal of constructs is permitted by the fact that removal of a construct header statement results in the removal of the entire construct. For example, removing a *try-header* statement will result in the removal of the *try block*, and, if present, the *catch clauses* and the *finally clause* of the *try* construct. Some inner parts of certain constructs are removable, such as an *else clause* in an `if`, a *case clause* in a `switch`, etc. Special methods in each enclosing construct, rather than statement list methods, must be used to remove such sub-constructs.

It is important to note that semantic consistency is not enforced. Semantic consistency is being considered as a future goal of the zJava compiler, although it is not clear at this time whether this can be achieved, and at what cost to efficiency. The complexity of semantic checks would add considerably to the execution time of a pass over HLIR. Furthermore, the nature of semantic checks is such that they would for the most part generate run-time, rather than compile-time, errors. For example, at compile time we can require that an expression is passed to the constructor of a `while` construct to represent the conditional expression. However, it is not always possible to verify that the expression evaluates to `boolean` until zJava actually executes. In remedy, the source code generated by zJava can easily be passed through a standard Java compiler, such as Sun's `javac`, which will identify semantic errors. It is considered reasonable to expect the user to write transformations which generate semantically correct code, as doing so only requires intimate knowledge of the Java language. *In contrast, user who wished to write syntactically correct transformations explicitly, in the absence of automatic consistency, would have to be familiar with all internal details of HLIR.*

2.2.2.2 Internal Consistency

The zJava HLIR is also designed to automatically update its internal structures when altered. For example, when statements are added to or removed from a statement list, the symbol table information is automatically updated, and types of symbols and expressions in the new statements are resolved. This relieves users of the need to understand the details of the scope and symbol table representation in HLIR, and of the need to maintain it correctly themselves. Additionally, HLIR disallows statement insertions and removals which would leave the IR in an inconsistent state. Examples of transformations which would otherwise break the internal structure are given when the statement list class is discussed in detail (§3.5.4).

The core functionality of an intermediate representation involves building objects to represent parts of a program (for example, statements), and using these to populate higher-level objects (for example, methods, and classes). In this kind of environment,

one common programmer error is object *sharing*, where the same object is used as a component of multiple IR objects. For example, several *symbol* objects, representing three different integer variables, might each have a reference to a *type* object representing the integer type. Should all of these *symbol* objects reference the same *type* object, a programmer who wished to change the type of one symbol might unwittingly change the type of all three. The concept of object *ownership*, as used in the Collections hierarchy of the Polaris project [5, 6], allows HLIR to to prevent such object sharing. When an object is inserted into a list, the list gains "ownership" of that object. An object can be owned by only zero or one list, and a list may not contain any objects it does not own. For example, attempting to add the same statement object to the body of a method more than once (or to the bodies of more than one method) will cause a run-time error. All objects in HLIR which are protected from sharing are either stored internally in such lists (which implicitly enforce ownership), or have their ownership explicitly controlled.

With object sharing disallowed, it is expected that users will frequently wish to create copies of objects. Each ownable object in HLIR provides a safe `clone` method, which produces a deep copy of the object (cloning all HLIR objects it owns). Further, we provide a *reference list* implementation, which allows objects to contain references to objects owned by another entity.

2.2.3 Modularity of Analyses

HLIR has the ability to save information from a compiler analysis for use by later passes. This is intended to allow one analysis to build on the results of another without repeating it. Possible solutions include the use of object serialization, outputting an enriched `.class` file format, or storing information in the generated source code.

2.2.3.1 Serialization

Java 1.1 introduced support for object serialization, which allows Java objects to be written to disk, and later read and used to regenerate the objects. This provides one possible way of saving the results of a compiler pass for later use. A potential

drawback is that the generated files could be unreasonably large. The use of the Java `transient` modifier can alleviate this by controlling which parts of an object can be discarded when it is serialized. Currently, the zJava HLIR does not make use of object serialization, although it would be straightforward to implement this in the future.

2.2.3.2 Compiler Directives

The zJava HLIR is designed primarily for source-to-source transformations. This suggests that the generated source code could be a place to store information from compiler analyses. The zJava HLIR includes support for passing user-defined directives to the compiler in the form of special comments in the source code, and for including annotations in the output source code. This technique is best suited to storing the results of an analysis (e.g. "this loop is parallel"), rather than the actual data structures generated and populated by the analysis (e.g., dependence graphs). While the latter would be possible, it could result in unacceptably large blocks of comments in the source code.

The Java grammar has been extended to include a syntax for specifying directives (see §4.8), and the framework has been established to simplify the task of adding additional directives in the future. Directives take the form of special comments in the source code, on the line(s) before the construct with which they are to be associated. During HLIR population, each directive is encapsulated inside a an attribute object and attached to the list of attributes on the appropriate HLIR object. Similarly, directives can be added to existing IR objects, and will be included in the generated source code at the appropriate locations. Currently, directives can be attached to a compilation unit, class, method, class/instance variable declaration, local variable declaration, or to any statement.

2.2.3.3 Rich Class Format

The `.class` file provides another possible location for storing analysis information. The `.class` file specification [7] includes support for user-defined attributes, which

are ignored by standard classloaders. Hence, it is possible to develop a "rich" class format to store the results of compiler analyses in the form of special attributes. The zJava Bytecode Intermediate Representation (BCIR) [8], which can read and write custom attributes, is used by HLIR to load classes external to the program being compiled. Bytecode generation [9] can store the results of analyses by including custom attributes while generating BCIR.

This technique is very useful for classes which are not generally available in source form. For example, Java class libraries could be analysed once, and compiled into augmented class files. Later analyses of user code which calls into these libraries would thus have access to the results of analysing the library classes, even if the source code was no longer available.

2.2.4 Control Flow Representation

The zJava compiler includes a representation of control flow, built on top of HLIR. A control flow graph (CFG), is constructed for each method, consisting of basic blocks linked together by flow information [10]. The CFG provides the necessary mechanisms for intraprocedural analysis, and contains all information necessary to implement interprocedural analyses.

The complex control flow involved in the Java exception model [11] introduces many problems when designing and constructing a control flow representation. The general problem is that the granularity of basic blocks is significantly affected by the presence of *potential exception-causing instructions* [12]. Any statement which *could* cause an exception (array access, pointer dereference, method call, etc) causes the end of a basic block. We have based our CFG model on the Jalapeno FCFG [12] from IBM, designed to reduce the impact of exceptions, while accurately modelling the resulting control flow.

2.2.5 Symbol Tables

Representing all information available in the source code involves capturing scope information. Thus, the scope of symbols must be represented in the IR.

Since Java allows local variable declarations anywhere in a block of statements, the task of determining and representing which symbols are visible at any point in the program is complicated. Normally, a new scope would be created at the beginning of each class, method, and block of statements. The close of these scopes is easy to determine—the end of the class, method or block. With the statements that cause the creation and close of each scope easy to identify, it is easy to populate the symbol tables while traversing the AST. However, in Java (and indeed in any language that allows local variable declarations anywhere in a block), every declaration statement must also be considered to begin a new scope. The difficulty arises when trying to determine the close of these local variable scopes—there is no matching "end" statement to identify it. Such scopes are considered to run from the declaration until the end of the enclosing statement block. Thus, any statement that ends a block must be considered to terminate one or more scopes.

Local symbol declarations can also exist in the *for-init* part of a `for` construct, the header of a *catch clause*, or the use of a *label* statement. The specific scope of each of these declarations is correctly modelled by the symbol table representation.

The zJava HLIR creates a symbol table for each unique scope, and chains each scope to its parent (the enclosing scope). Since Java scopes are only nested within one another, zJava symbol tables are linked in a tree structure. The resulting symbol table tree is constructed in conjunction with HLIR, and is automatically updated whenever symbol information changes (modifying the statement list of a method body, adding or removing a class variable, etc). Symbol lookups automatically proceed up the tree until the highest level scope is encountered. This design implicitly implements variable hiding; the closest declaration of a variable is seen first, so local variables can shadow instance variables. Every statement represented in HLIR includes a link to the symbol table for the scope in which it resides. Thus, when examining any statement in the program, HLIR presents what appears to be a single symbol table

containing all symbols visible to that statement. Qualified symbols (e.g. `MyClass.a`) are resolved by querying the symbol table in the class of the qualifying type (class `MyClass`).

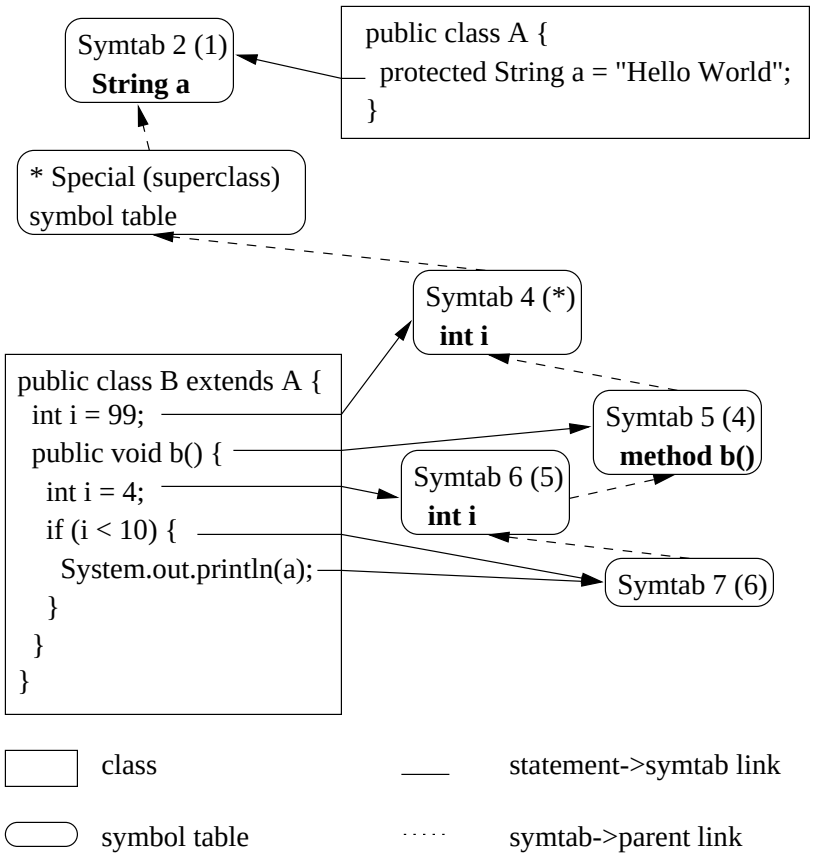


Figure 2.1: Symbol table tree example, showing inheritance and hiding.

Figure 2.1 shows a Java code example, and the HLIR symbol table tree which would result. Symbol tables are labelled with the symbol table number followed by the number, of its parent in parentheses. The symbols declared within each scope are listed (in bold) inside the symbol table. In this figure, the use of the symbol `i` in the method `b()` is resolved to the declaration in symbol table 6, which hides the earlier declaration (in symbol table 4).

In addition, the fact that some superclass symbols may be visible complicates the representation of scoping, and the implementation of symbol table lookups. When a local (unqualified) symbol is used inside a method, it is not sufficient to simply traverse the symbol table representation upwards until the class level is reached,

as this would ignore symbols inherited from the superclass. HLIR symbol table lookups are designed to automatically examine available superclasses, loading them from `.class` files if necessary. In Figure 2.1, the symbol `a` declared in class `A` is inherited by class `B`, and thus available to all statements in method `b()`.

2.2.6 Implicit Execution

One unique aspect of modelling Java code is the fact that a Java compiler must generate some code implicitly, in ways which are not necessarily obvious from the source [4]. The zJava HLIR implicitly adds statements to represent this where appropriate. These statements do not correspond to actual source lines, and as such are not normally included when HLIR is converted to source code. The zJava compiler can optionally be instructed to generate (commented out) implicit code during source code generation.

Every Java compilation unit is implicitly assumed to import the package `java.lang`. If such an import is not explicitly declared, HLIR automatically adds the declaration `import java.lang.*`. If a class does not specify a superclass (and is not itself `java.lang.Object`), HLIR implicitly alters the class to extend `java.lang.Object`. These changes relieve HLIR methods of the need for special cases when referencing import and superclass declarations.

Any class, or class member declaration which does not include an explicit access modifier (`public`, `private`, etc) is assumed to have the default (or *"package"*) access modifier, which has no source code equivalent.

Implicit assignments made by the virtual machine (and not the compiler) are not represented in HLIR. These include assignment of default values to class fields and array members [4, §4.5.4], and assignment of exceptions to catch exception variables [4, §14.1].

2.2.6.1 Variable Initializers

The most straightforward example of implicit execution arises when a variable is assigned some initial value in the declaration. Internally, declaration statements are

not part of the representation. Instead, each new declaration causes the creation of a new scope, and the symbol declared is added to the symbol table for this scope. If the declaration included an initializer, an implicit statement is inserted to represent the assignment of this initial value to the variable. This statement is inserted in order to accurately model the execution of the initialization code. If new local variables declarations, with initializers, are added to an existing method body, the associated implicit statement is automatically generated.

It is important to note that variable initializers are also included in the appropriate symbol table entries. All symbols in the table have a flag indicating whether or not they include an initializer, and methods are provided to access them. We choose to also represent the execution of the assignment in the statement list, to allow a simple iteration over a method body to see all executed statements (without having to check the symbol table at each point).

2.2.6.2 Implicit Constructor Invocation

A class constructor may, as the first statement in its body, explicitly invoke either a superclass constructor, or another of its own constructors. Any constructor which does not do either of these is assumed to implicitly call the parameterless superclass constructor. Thus, at times we insert a call to `super()` as the first statement in the body of a constructor (line 17 in Figure 2.2 on page 20).

2.2.6.3 Instance Variables and Instance Initializer Blocks

If the class declares any *instance initializer blocks* (lines 4–6 in Figure 2.2 on page 20), these are implicitly executed every time a new instance of the class is created. These blocks are executed immediately following the superclass constructor invocation. If the class declares more than one instance initializer block, they are executed in the textual order in which they appear in the source code. In HLIR, instance initializer blocks are placed in a special method called `<object_init>()` (line 32 in Figure 2.3), and an implicit call to this method is added in every appropriate constructor (line 18 in Figure 2.2), immediately following the superclass constructor invocation.

If a class declares any instance (non-static) variables with initializers (line 2 in Figure 2.2), the initialization of these variables must also be implicitly represented. This initialization occurs after the superclass constructor invocation, and before the execution of any instance initializer blocks. This is represented in the zJava HLIR by inserting a block of implicit statements, one to represent the initialization of each instance variable, at the beginning of the special method `<object_init>()`. This is demonstrated by line 31 in Figure 2.3, where the initialization of the instance variable `baz` is included in `<object_init>()`.

When new instance variables, with initializers, are added to an existing class, the associated implicit statement is automatically generated and added to `<object_init>()`.

During the development of HLIR, we determined empirically (using BCIR to examine the bytecode generated by `javac`) that instance variable initializers and instance initializer blocks are executed exactly once each time a new class instance is created. That is, a constructor which begins with a explicit call to another constructor of its class (line 24 in Figure 2.2) does not execute `<object_init>()` (because the constructor it invokes will do so). A clarification in the second edition of the Java Language Specification [4, 12.5] confirms these results.

2.2.6.4 Class Variables and Class Initializer Blocks

Class initializer blocks (static initializer blocks) must be handled differently. These blocks are executed by the JVM when a class is initialized, which occurs at the first *active use* of the class [4, §12.4.1]. Such blocks are grouped into the special method `<clinit>()` (lines 35–39 in Figure 2.3), but the execution of this method is not represented. The most conservative representation would involve placing a call to `<clinit>()` before each active use of a class, with guard code to see if the class has already been initialized. This would significantly increase the size of the representation and generate false dependencies, reducing the effectiveness of many possible analyses. Further analysis could be performed, using the zJava HLIR, to more efficiently represent the execution of `<clinit>()`. This would require an inter-

```
0:    public class Foo {
1:      // An instance variable, with initializer:
2:      private String baz = new String("baz.");
3:
4:      { // an instance initializer block
5:        System.out.println("new Foo!");
6:      }
7:
8:      // A class variable, with initializer:
9:      private static String staticBaz = new String("staticBaz.");
10:
11:     static { // A class initializer block
12:       System.out.println("class Foo initialized");
13:     }
14:
15:     // A constructor with no explicit constructor invocation:
16:     public Foo() {
17:       super();
18:       <object_init>();
19:       // constructor body
20:     }
21:
22:     // A constructor with an explicit constructor invocation:
23:     public Foo(int i) {
24:       this();             // explicit in the original source code
25:       // nothing implicitly added
26:       // constructor body
27:     }
28:       // Note, lines 29--40 are shown in Figure 2.3.
41:   }
```

Figure 2.2: Examples of implicit code added in to a class. Implicitly added code is emphasized.

procedural analysis, and is beyond the scope of this thesis. We chose to simply group these initializers in `<clinit>()`.

The execution of class (static) variable initialization (line 9 in Figure 2.2) is similar to that of instance variables. Class variable initializers are executed just before class initializer blocks. This is represented in the IR by inserting an implicit statement for each class variable initializer into a block at the beginning of the special method `<clinit>()` (line 37 in Figure 2.3). Class initializer blocks (lines 11-13 in Figure 2.2)

```
29:     // The implicit method <object_init>():
30:     private void <object_init>() {
31:       { baz = new String("baz."); }
32:       { System.out.println("new Foo!"); }
33:     }
34:
35:     // The implicit method <clinit>():
36:     private void <object_init>() {
37:       { staticBaz = new String("staticBaz."); }
38:       { System.out.println("class Foo initialized"); }
39:     }
40:
```

Figure 2.3: The implicit initialization methods generated for class Foo from Figure 2.2.

are then added to `<clinit>()` (line 38 in Figure 2.3).

If any new class variables, with initializers, are added to an existing class, the associated implicit statement is automatically generated and added to `<clinit>()`.

2.2.6.5 Implicit Constructor Declaration

If a class defines no constructors, a default constructor must be added. This constructor takes no parameters and simply includes an implicit call to `super()` (unless the class is `java.lang.Object`), and a call to `<object_init>()`. The default constructor is declared `public` if the class is also public, otherwise it is given the default access modifier.

2.2.7 External Class Loading

Interprocedural analyses often require information about classes for which the source code is not available. For example, some Java libraries are usually only available in compiled (`.class` file) form. The zJava HLIR includes the ability to construct the intermediate representation from `.class` files generated by any compiler. Using either BCIR or a standard classloader, HLIR automatically locates and loads `.class` files when needed (e.g. during symbol lookup). The resulting representation includes

all information available in the `.class` file, except for the actual bodies of methods, as there is no attempt to decompile bytecodes.

It is important to note that the name of the class to be loaded is determined statically (at compile-time). Thus, it is possible that the class loaded by HLIR will not be the same as that which would be loaded if the code being examined were run on a JVM. For example, HLIR will determine that the return type of the method `java.lang.Class.newInstance()` is `java.lang.Object`, and will load `Object.class`. However, the `newInstance()` method will, at runtime, return a new instance of the class represented by the `java.lang.Class` object on which it is invoked.

2.2.8 Inner Classes

Then Java 1.1 specification [13] includes, among other things, the addition of nested classes [14, 15]. This introduces several issues which a Java IR must address. The representation of classes must be extended to mark those which are inner classes, and to allow classes to contain other classes. Similarly, the representation of methods must be extended to allow methods to contain classes. Inner classes in Java have access to instance variables declared in the enclosing class (and local variables in the enclosing method, if the inner class is declared within a block).

The HLIR representation of scopes and symbol information will support this type of scoping with little or no change. As well, the HLIR representations of classes and methods can be easily extended to include the additional information required to support nested classes.

The zJava HLIR does not support inner classes, and they will be ignored in input source code. HLIR has been designed with the future support of nested classes in mind; situations where additional work will be needed are identified. We currently raise an exception if a user creates an inner class construct (such as `ClassName.this`) using HLIR methods, and attempts to insert it into an existing class.

2.2.9 Source Code Generation

The zJava HLIR includes the facility to convert the IR to source code. The generated code is intended to be human-readable, with appropriate use of indentation and spacing. Source code generation is intended to be invoked at the compilation unit level; several complicated issues involved are handled automatically (see §3.6.4).

2.2.10 Attributes

HLIR includes the ability to attach a list of user-defined attributes to any object in the representation. A base class is provided, from which subclasses can be derived containing any information one may wish to associate with an HLIR object. For example, compiler directives are attached as a special class of attribute on objects in the IR.

2.3 Related Work

In this section we discuss existing frameworks for program representation and manipulation. We first discuss high-level systems, then examine several existing systems for the manipulation of Java programs at the bytecode level.

2.3.1 High-Level Frameworks

The Polaris[2] [1] Fortran compiler, from the University of Illinois at Urbana-Champaign (UIUC), is a production-quality tool for experimentation with new transformation techniques for parallelization. Polaris implements the concept of object ownership though its Collections hierarchy. The Polaris internal representation [5, 6] uses an AST representation to provide a robust framework for source-to-source transformations. The syntactic correctness of the represented Fortan program guaranteed at all times, and the structures of the IR (including control flow information) are incrementally updated under transformations.

[2] http://polaris.cs.uiuc.edu/polaris/polaris.html

The SUIF1.0[3] [16] compiler system, from Stanford University, provides a framework for experimental research on new compiler techniques. SUIF1.0 consists of a variety of tools and optimization passes, all operating on a common intermediate representation. The intermediate representation of SUIF is targeted to imperative languages (Fortran, C), and does not include support for object-oriented language constructs.

SUIF2.0[4] [2] is a new implementation of SUIF, designed to facilitate extension of the intermediate representation. Unlike SUIF1.0, different modules in SUIF2.0 can act together (rather than through intermediate files). The form of the IR is the same as SUIF1.0, composed of a low-level AST representation augmented by high-level structures. However, an object-oriented design is used, intended to facilitate the addition of new classes to represent languages other than C and Fortran. Also, machine-level optimizations are more extensive than those permitted by SUIF1.0. An example of such an extension is OSUIF[5] [17, 18], which introduces support for the representation of object-oriented languages, allowing for the prototyping of object-oriented optimizations. OSUIF extends the SUIF2.0 symbol table to represent subclass and subtype relationships, the IR to model dynamic dispatch, and the CFG to model exceptions.

Score[6] [19] is a compiler for heterogeneous parallel systems, developed at the University of Massachusetts. The Score IR is designed with four major goals in mind: permitting arbitrary ordering of transformations; representing diverse architectural features; facilitating the extension of the IR to cover new architectural features; and encouraging reuse of transformations and IRs for different architectures. The representation is a variant of the program dependence graph [20]. The low level single static assignment [21] (SSA) form is augmented by a high-level representation of certain important constructs. The two representations are maintained in parallel, allowing low- and high-level transformations to be interleaved.

[3] http://suif.stanford.edu/suif/

[4] http://suif.stanford.edu/suif/suif2/

[5] http://www.cs.ucsb.edu/~osuif/

[6] http://celestial.cs.umass.edu/McKinley/ir.html

The Sage++[7] [22, 23] toolkit, from Indiana University, provides an object-oriented framework for building Fortran, C, and C++ program restructuring systems. The system consists of a set of parsers which populate a structured parse tree, a symbol type table, and a set of source code annotations. Sage++ parses source code into an intermediate file format, and converts this back to source code after transformations have been applied. Control flow and data dependence representations are built on the code representation.

The Paraphrase-2[8] [3] automatic parallelizing compiler, from UIUC, is a research tool for experimenting with program transformations. Frontends for C and Fortran parse into an IR composed of a data dependence graph (DDG), a control flow graph, and a call graph. The IR can be converted back to source code. A debugger is provided to examine the internal structures, and a graphical interface allows viewing and modification of the DDG.

The Illinois Concert[9] [24, 25] system, also from UIUC, is an optimizing compiler for ICC++, a concurrent object-oriented programming model. The frontends target a core intermediate representation, which is then transformed, via a CFG, into an SSA program dependence graph, upon which optimizing transformations are applied. The PDG is converted back to a CFG, and finally to RTL [26], where register allocation and machine code generation are performed.

The Vortex[10] [27] compiler infrastructure is a language-independent optimizing compiler for object oriented languages. Developed at the University of Washington, Vortex has frontends for Cecil, C++, Java bytecode, and Modula-3. These interface with a common backend, through the intermediate representation. The IR is a three-address, low-level form, with high-level constructs for certain operations (message sends, field accesses, runtime type tests, object creation). The high-level constructs are converted to low-level constructs during the optimization phase.

FrIL [28] is an intermediate language with *fractal* properties; that is, the same

[7] http://www.extreme.indiana.edu/sage/index.html

[8] http://sparta.csrd.uiuc.edu/parafrase2/

[9] http://www-csag.ucsd.edu/projects/concert.html

[10] www.cs.washington.edu/research/projects/cecil

program transformations can occur at different levels in the representation. Frontends parse C or Fortran into FrIL, where optimizing transforms are applied. During optimization, the front end may be asked to lower parts of the representation, converting operations into a lower form for further optimization.

The Optimizing Oberon-2 Compiler[11] (OOC2) uses a combined intermediate form, generated in a single pass over the source program. The guarded single-assignment [29] (GSA) form combines high-level control structures (loops, structured data accesses, if-statements, etc) with machine-level instruction lists and a static data flow graph, within one representation.

FLINT[12] [30, 31, 32], from Yale University, is a typed common intermediate form which allows for the modelling of the semantics and interactions of several high-order, typed (HOT) languages. The IR is based on polymorphic lambda calculus, and supports conventional dataflow and loop optimizations, as well as lambda calculus-based operations. The FLINT backend generates machine code from the common format, allowing HOT languages to share a code generator and runtime system. Currently, FLINT supports ML and a subset of Java; frontends for "Safe C", Haskell, Java, and other HOT languages are under development.

Table 2.1 summarizes the functionality provided by several of the systems discussed above, in addition to zJava. The comparison criteria are: automatic syntactic consistency enforcement under transformation; support for modular analysis (where one compiler pass can build on the results of another at a later time); the ability to convert the IR into source code; representation of both high- and low-level constructs; mechanisms to support interprocedural analysis; and the languages supported by the compiler.

2.3.2 Bytecode IRs

Several Java optimization and native code generation projects operate on Java bytecodes, rather than source. These frameworks are in some ways complimentary to

[11] http://www.oberon.ethz.ch/

[12] http://flint.cs.yale.edu/

	Automatic Consistency	Modular Analysis	Source-to-Source	High and Low Level	Support for Interprocedural	Languages Supported
zJava HLIR	●	●	●	○	●	Java
SUIF[1]	○	●	●	●	●	Fortran, C, C++
Polaris	●	●	●	●	○	Fortran
Score	○	○	○	●	○	N/A
Paraphrase-2	○	○	●	○	○	Fortran, C
Concert	○	●	○	○	●	IC++
OOC2	○	○	○	●	○	Oberon-2
FrIL	○	○	○	●	○	Fortran, C
FLINT	○	○	○	○	○	ML, Java, "Safe C", Haskell
Sage++	○	●	●	○	○	Fortran, C, C++
Vortex	○	○	○	●	●	Cecil, C+, Modula-3, Java bytecode

Table 2.1: Comparison of various high-level compiler frameworks. A filled circle represents that the system has the associated functionality.

[1]We include SUIF1.0, SUIF2.0, and OSUIF

HLIR; code transformed by HLIR can easily be passed through a bytecode optimizer for low-level optimization.

While certain optimizations are better suited to bytecode representations, there are drawbacks to using bytecode, rather than Java source, as the input:

- expensive type-elaborating flow analysis required (bytecode operations are not typed)
- information about certain high-level constructs is lost, including inner classes, finalization, and some synchronization primitives, which are expanded into a low-level representation in bytecode [33]
- the `.class` file format is designed for compactness, and is awkward to work with directly
- expressions are not explicit in bytecode, and can span an arbitrary number

of opcodes (which may even be in different basic blocks), making even simple transformations difficult

For these reasons, several projects elect to define their own IR, rather than operating on bytecode directly.

The Marmot [33] optimizing native static Java compiler, from Microsoft Research, first translates bytecode into JIR, a temporary variable-based, strongly typed, three-address IR, in SSA form. Standard optimizations, object-oriented optimizations, and Java-specific optimizations are performed on this IR, which is then translated into a machine-level IR for native code generation. Optimizing transformations are expected to preserve type correctness themselves.

The Soot [34, 35] project at McGill University aims to provide a suite of tools to simplify bytecode optimization. The core of this framework is the Jimple bytecode IR [36], a three-address representation which replaces the bytecode stack with typed local variables. Use of the Soot framework involves translating bytecode unto the Baf representation, then to Jimple (where optimizing transforms are applied), then to Grimp, and finally back to bytecode. The Soot project is also investigating the usefulness of using custom attributes in `.class` files to store the results of analyses.

The Jalapeno Virtual Machine [37], from IBM Research, takes a compile-only approach to program execution. The Jalapeno dynamic optimizing compiler [38] actually consists of three compilers: a baseline compiler, a quick compiler, and an optimizing compiler. At runtime, the VM chooses one of the three to compile each method to native code. The optimizing compiler translates bytecodes into HIR, a register-based representation where instructions are represented as n-tuples, and arranged in basic blocks. Optimization passes are performed on HIR, which is then lowered into LIR for Jalapeno-specific operations, and finally into MIR for native code generation.

The gcj compiler [39] from Cygnus Solutions is a Java frontend to the GNU GCC compiler. Bytecode is translated into a tree-level representation with typed expressions. This is then lowered into the RTL [26] format, for machine-level optimization and code generation.

The Caffeine [40] prototype Java bytecode to native compiler has been built on top of the IMPACT [41] compilation infrastructure at the University of Illinois at Urbana-Champaign. Caffeine first translates bytecode into Java IR, then into LCode, the machine-level intermediate form of the IMPACT infrastructure. Optimizations and code generation are then performed on LCode.

The j2s [42] system provides a Java frontend to SUIF1.0, translating Java bytecodes into SUIF, and using annotations to represent specifically object-oriented aspects. j2s does not include support for exceptions or thread synchronization, and requires that all `.class` files of the program be present at compile time, and thus does not support dynamic linking. The SUIF library is used to implement optimizations, and the resulting native code is interfaced with the JVM through either the Java Native Interface or the Just-In-Time API.

A Java bytecode frontend to both SUIF2.0 and OSUIF, also called j2s [43], is designed for static whole-program compilation. Again, exception handling, threads, and dynamic loading of classes are not supported.

Finally, the zJava ByteCode Intermediate Representation [8] (BCIR) provides a framework for the manipulation of Java bytecodes. Through the use of compiler directives and custom bytecode attributes, HLIR and BCIR can work together to carry optimizations from the high-level through to the lower level.

Chapter 3

HLIR Classes

In this chapter we give details of the core classes of HLIR. We begin with the top level of the hierarchy, the representation of an entire program. In the subsequent sections, we describe the representation of progressively lower levels, including compilation units, classes, methods, statements, expressions, and symbols. In addition, the classes used to represent types, access modifiers, and symbol tables are presented.

3.1 Program

The zJava compiler is invoked by specifying, on the command line, the set of Java source files (compilation units) to load. This set of files is referred to as a single program. The **Program** class is the root of the representation, and represents this set of source files. It contains methods for populating a **Program** object with source files, and for iterating over the compilation units of the program.

3.2 Compilation Unit

The **compilationUnit** class represents a single Java source file. It includes the name of the source file, the name of the package it belongs to, and an optional list of import declarations. The **compilationUnit** also contains a representation of each class or interface defined in the source file. Methods are provided to access all of

this information, to search for a particular class (and obtain an HLIR representation of it), and to output the **compilationUnit** in either a debug or source code form. Several of the methods of class **compilationUnit** are summarized in Table C.1 of Appendix C (page 117).

3.2.1 External Class Resolution

During HLIR population and the execution of user compiler passes, it may be necessary to resolve symbols from classes external to the **Program** (e.g., to determine the type of `System.out.println()`). HLIR automatically locates and loads compiled external classes when necessary.

Methods are provided in **Program** and **compilationUnit** to search all sources for a class, and return and HLIR representation of the class. Given the name of a target class, and the *package context*[1] (§3.2.1.2) in which to search, these methods search all sources in the appropriate sequence (§3.2.1.1), looking for a matching class. If a matching class already exists in the **Program**, its HLIR representation is returned. If the matching class is found externally, in compiled form, the class is loaded using **zClassLoader**[2] (§4.6), which returns a `java.lang.Class` reflection. The equivalent HLIR is then constructed from this reflection, containing all information except the actual opcodes (body statements) of methods.

The external class resolution method in **Program** searches all sources available to the program for a matching class. This method first checks the **compilationUnits** of the **Program**, then the set of paths defined by the environment variable `CLASSPATH`, for an accessible class matching the search target.

If a matching class is found, an object is constructed to represent it. To facilitate later searches for the same class, the newly constructed object is cached in a special compilation unit, <`dyn`>, contained in the **Program**, and a symbol is added to that compilation unit's symbol table.

[1] The package of the class containing the statement that references the target class.

[2] Note, HLIR can also make use of BCIR to load classes.

3.2.1.1 Search Sequence

The method `findAccessibleClass` in class **compilationUnit** attempts to resolve external classes by following the sequence outlined in the JLS [4]:

- Search the **compilationUnit** for a matching class.
- If the target is specified in any *single-type-import* declarations (e.g. `import java.util.ListIterator`), attempt to load it from the `CLASSPATH`.
- Search globally for a matching class in the same package context, using the method provided in **Program**.
- If the compilation unit specifies any *import-on-demand* declarations (e.g. `import java.io.*`), search the `CLASSPATH` for a matching class in each package imported.

If an accessible matching class is found, an HLIR object representing it is constructed and cached in the **compilationUnit**. A symbol for the new class is added to the compilation unit's symbol table (to facilitate later searches for the same class).

3.2.1.2 Class Accessibility

The accessibility of a target class depends on the package context in which it is used. For example, in the compilation unit shown in Figure 3.1, a search for the class `IOException` would occur from the package context of package `mypackage`. Note that compilation units with no `package` declaration all belong to the same nameless package, "".

Accessibility of a class requires that:

- the class is not `private`
- the class belongs to an accessible package
- the class belongs to the same package as the package context

```
package mypackage;
import java.io.*;

class myClass {
  public void myMethod(OutputStreamWriter stream)
    throws IOException
  {
    BufferedWriter bf = new BufferedWriter(stream));
    bf.write("Hello World.");
  }
}
```

Figure 3.1: A simple compilation unit.

Accessibility of a package is system dependent, but usually means that it is in the set of paths defined by the environment variable `CLASSPATH`, and it is readable. Based on these requirements, if the `import` declaration were to be removed from the example in Figure 3.1, then the class `IOException` would only be accessible if it belonged to the package `mypackage`.

Note that the `CLASSPATH` searched is that of the zJava compiler's execution context, which may not be the same as the context in which the program being manipulated will eventually execute. It is straightforward to add a command line option to the compiler to control this variable, should it become necessary in the future.

3.3 Classes

A **ClassShell** object represents all information about a single Java class or interface. **ClassShell** contains information available from a `.class` file, not including actual opcodes (the bytecodes are not decompiled into Java statements). A symbol table is included, containing symbols for the class fields and methods. **ClassShell** provides access to pertinent information such as the name of the class, the name of its superclass, and the access permissions of the class. Each **ClassShell** includes a reference to its entry in the enclosing compilation unit's symbol table. Functionality is also provided to iterate over the methods defined in the class, and to add or remove import

declarations, methods, fields, and initializer blocks. A **ClassShell**, which was built from compiled code, cannot be converted back to either source code or the `.class` file format. Table C.2 of Appendix C (page 118) summarizes several of the methods of **ClassShell**.

The class **classObject** extends **ClassShell** to include information present in the source code, such as the bodies of each method (statements, expressions, and symbols). The **classObject** class also provides a method to convert the class to Java source. Two special methods are implicitly added to every class: `<object_init>()` and `<clinit>()`. The `<object_init>()` method is used to represent the execution of both instance variable initializers and instance initializer blocks. Similarly, the `<clinit>()` method represents class variable initializers, and class initializer blocks. These are arranged with the variable initializers first, followed by the initializer blocks, in the textual order in which they appeared in the source code [4]. When new initializers are added, they are added to the end of the appropriate implicit method. Similarly, if a field with an initializer is added, it is added after all existing fields. During source code generation, these methods are converted into class and instance initializer blocks.

3.4 Methods

The class **MethodShell** represents methods constructed from compiled code, which do not include the body statements. **MethodShell** provides methods to access and modify information such as the name of the method, its access permissions, return type, formal arguments, etc. Unique method *signatures* (used in symbol table entries) for each method are formed by concatenating the method name and the type signature (§3.10) for each formal argument, delimited by "$" symbols. Since two methods with the same name, argument types, and argument order cannot have the same return type, there is no need to include the return type in the signature. It is important to note that constructors are renamed to `<init>` in HLIR, making it possible to differentiate a constructor from a method whose name is the same as its class, and

whose return type is the type of its class, and which has the same argument types as a constructor of the class. Each **MethodShell** includes a reference to its entry in the enclosing **ClassShell**'s symbol table.

The **methodObject** class extends **MethodShell** to include the body statements of the method, and the ability to convert the representation to Java source. Special methods convert the bodies of the implicit `<object_init>()` and `<clinit>()` methods into class and instance initializer blocks. Table C.3 of Appendix C (page 118) summarizes several of the methods of **methodObject**.

The representation of methods in a **ClassShell** is in the form of **MethodShell** objects, while methods in a **classObject** are represented by **methodObjects**.

3.5 Statements

In this section, we discuss the hierarchy of classes used to represent and arrange Java statements. We begin with a discussion of the base class, then describe the various derived classes used represent each type of statement. The statement list class is then discussed, followed by the details of how individual statements are arranged into statement lists to form Java constructs and method bodies. Throughout this section, we discuss how each component of the statement class hierarchy contributes to consistency enforcement.

Figure 3.2 details the hierarchy of classes used to represent and arrange Java statements. All classes representing statements derive from the base class **zStatement**. This base class is derived from class **zObject**, which will be described in §4.3.

The base class **zStatement** includes methods to examine the symbol table for the scope in which the statement resides, to iterate over all expressions contained in the statement, to obtain a reference to the statement list containing the statement, to convert the statement to source code, and to obtain the unique (across an entire **Program**) tag of the statement. Each class derived from **zStatement** also has an associated unique type, which can be accessed through a method in the base class. Table C.4 of Appendix C (page 119) summarizes several of the methods of

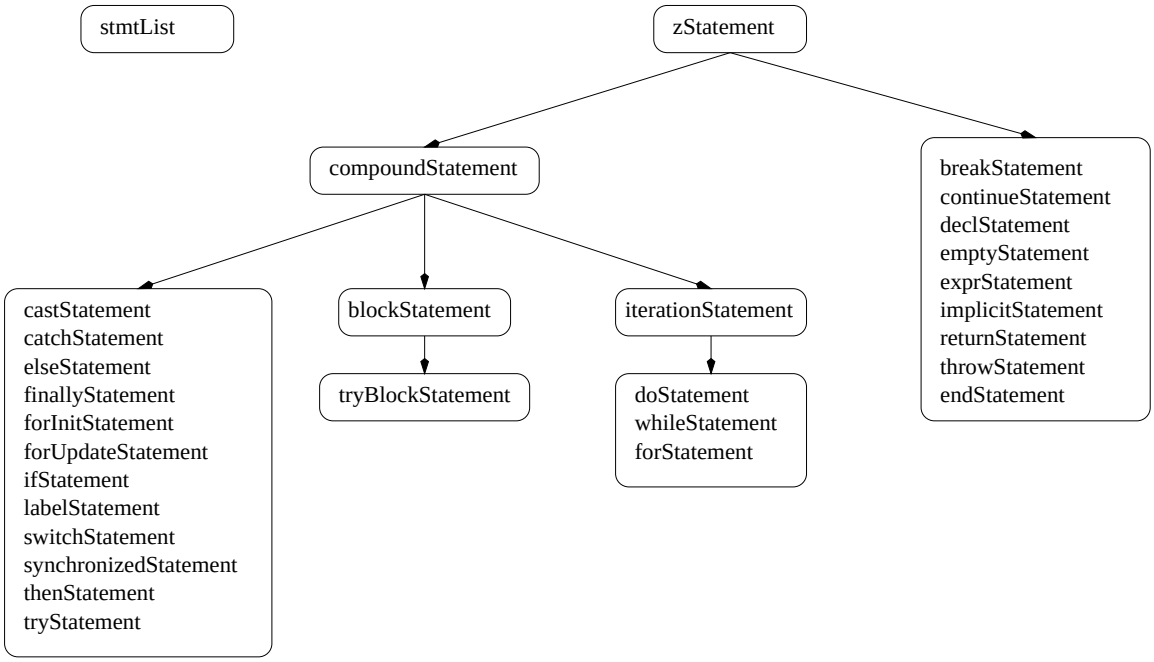


Figure 3.2: Class hierarchy of the zStatement package.

zStatement.

A class is derived for each type of Java statement, containing additional fields and methods required to represent it. The constructors of each class derived from **zStatement** are designed to prevent the creation of incomplete or syntactically incorrect statements. For example, creating a statement to represent a new *catch clause* header requires that the *catch exception variable* be specified.

In an AST representation (and, in the Java grammar [4, §19]), several statements contain other statements. For example, an `if` statement is considered to contain a statement for the *then-part*. This statement may itself contain others—for example, if it is a *block*. The statement representation in HLIR is non-recursive; that is, statements contain only expressions and symbols, not other statements. Statements are stored in "flat" statement lists, each of which is contained in an object of class **stmtList**. Statements which contain other statements in an AST representation are represented in HLIR as a sequential list, starting with a *header* statement, and ending with an *end* statement. We refer to these "flattened" statements as *constructs*.

The majority of syntactic consistency enforcement in HLIR is handled by three

zStatement class	Java equivalent	JLS Section
emptyStatement	";"	[4, §14.5]
breakStatement	"`break`" <*label*> ";"	[4, §14.13]
continueStatement	"`continue`" <*label*> ";"	[4, §14.14]
returnStatement	"`return`" <*expression*> ";"	[4, §14.15]
throwStatement	"`throw`" <*expression*> ";"	[4, §14.16]
exprStatement	<*expression*> ";"	[4, §14.7]
declStatement	see §3.5.1.4	[4, §14.3]
implicitStatement	see §3.5.1.3	none

Table 3.1: Single statements (not part of a construct), with references to JLS [4] sections.

mechanisms:

1. *zStatement constructors.* The constructors of each class derived from **zStatement** ensure that malformed or partially-formed statements (such as a **whileStatement** with no conditional expression) cannot exist in the representation. These will be described in detail in §3.5.1, §3.5.2 and §3.5.3.

2. *stmtList constructors.* The constructors of class **stmtList** prevent the creation of a statement list containing a malformed or partially-formed construct (such as a `while` loop with no body statements). These will be described in §3.5.4.

3. *stmtList modification methods.* Methods in class **stmtList** which allow modification of the list (insertion or removal of statements) ensure that the modified list is syntactically correct. This will be described in §3.5.5.

3.5.1 Single Statements

Various **zStatement** classes exist to represent statements which are not part of a construct. These statements are summarized in Table 3.1, including an example of their source code equivalents, and a reference to the appropriate section in the Java Language Specification [4]. The following sections explain the details of certain single statements.

3.5.1.1 Expression Statement

The class **exprStatement** represents an expression statement [4, §14.7], containing of a single **zExpression** (§3.6) object (which may contain subexpressions). The constructor of this class requires that a non-null expression object be provided.

3.5.1.2 Break and Continue Statements

Break and continue statements [4, §14.13, §14.14] are represented by the **breakStatement** and **continueStatement** classes. Each contains an (optional) label symbol, which refers to the symbol table entry for the branch target.

3.5.1.3 Implicit Statements

The **implicitStatement** class is similar to **exprStatement** except that it represents code implicitly added by HLIR, and has no source code equivalent. These statements are not included during source code generation. A command-line flag can be used to instruct the compiler to generate them (as comments) if desired. See §2.2.6 for details on statements implicitly added to the representation.

3.5.1.4 Declaration Statements

A local variable declaration statement, which declares and optionally initializes one or more symbols, is represented in a **declStatement** during HLIR population. However, these statements are not part of the final representation. Instead, when scope analysis is performed on each method body, the declared symbols are added to the appropriate symbol table, and the **declStatements** are removed. When a declaration includes an initializer, an **implicitStatement**, with a reference to the symbol table entry, is added at the point of the declaration to represent the initialization of the variable.

A **declStatement** will never appear in the statement list representing a method body. However, new local variables can be added to a method by creating and inserting an appropriate **declStatement**, which will be immediately converted into symbol table information by the list insertion methods. Variables can also be added

to a symbol table directly, but this will bypass the creation of implicit initializer statements for declarations with initializers. During source code generation, a declaration statement is created in the correct place for each symbol.

3.5.2 Compound Statements

The class **compoundStatement** is an abstract class derived from **zStatement**, and represents the *header* of a Java construct. Classes are derived from **compoundStatement** to represent the header of each kind of construct. The **compoundStatement** class includes a reference to the corresponding **endStatement**, which is used to represent the end of the construct. The **endStatement** contains a reference back to its corresponding **compoundStatement**. The following sections describe the various classes derived from **compoundStatement**.

3.5.2.1 Block Header

The class **blockStatement** is used to represent the header of a **block** [4, §14.2]. A block is any (possibly empty) list of statements enclosed in cursive brackets ("{ }").

3.5.2.2 If Construct Header

The header of an `if-then-else` [4, §14.8] (or `if-then`) construct is represented by the class **ifStatement**, which requires that a non-null expression object representing the conditional be provided to the constructor. The classes **thenStatement** and **elseStatement** are provided to identify the headers of the *then-part* and the *else-part*, respectively. The **ifStatement** class also includes a special method to handle the removal of the *else-part*.

3.5.2.3 Labelled Construct Header

Any Java construct can be preceded by a *label* [4, §14.6]. We indicate the presence of a label with the class **labelStatement**, and require that a non-null symbol (representing the name of the label) be provided to the constructor.

3.5.2.4 Switch Construct Header

The beginning of a `switch` [4, §14.9] construct is indicated by a **switchStatement**. The constructor of this class requires that a non-null expression object, representing the switch expression, be provided. The class **caseStatement** is used to indicate the beginning of a *case clause*, and requires that a non-null expression object, to represent the case expression, be provided to its construtor. The class **switchStatement** also includes a special method to handle the removal of a *case clause*.

3.5.2.5 Try-Catch-Finally Header

The beginning of a `try-catch-finally` [4, §14.18] (or `try-catch`, or `try-finally`) construct is indicated by a statement of class **tryStatement**. The header of the *try-block* is represented by a statement of class **tryBlockStatement** (which is derived from **blockStatement**). The header of a *catch clause* is indicated by a statement of class **catchStatement**. The constructor of class **catchStatement** requires that a non-null expression object, representing the *catch exception variable*, be provided. The class **finallyStatement** is used to indicate the beginning of a *finally clause*. The class **tryStatement** also includes special methods to allow for the removal of the *finally clause* or a *catch clause*.

3.5.2.6 Synchronized Block Header

The header of a `synchronized` block [4, §14.17] is indicated by a statement of class **synchronizedStatement**. The constructor of this class requires that a non-null expression object, representing the variable to be used for the lock, be provided.

3.5.3 Iteration Statements

Java contains three kinds of *iteration* statements: the `for` loop, the `while` loop, and the `do` loop. The abstract class **iterationStatement** extends **compoundStatement**, and is used to represent loop header statements. These include **forStatement**, **whileStatement**, and **doStatement**.

3.5.3.1 While Loop Header

The class **whileStatement** represents the header of a `while` loop [4, §14.10]. The **whileStatement** constructor requires that a non-null expression object be provided to represent the loop conditional.

3.5.3.2 Do Loop Header

The class **doStatement** represents the header of a `while` loop [4, §14.111]. The **doStatement** constructor requires that a non-null expression object be provided to represent the loop conditional.

3.5.3.3 For Loop Header

The header of a `for` loop [4, §14.12] is represented by the **forStatement** class, whose constructor requires that a non-null expression object be provided to represent the loop conditional. The header statement class **forInitStatement** is used to identify the beginning of the loop *initializer* [4, §14.12.1], and the class **forUpdateStatement** is provided to identify the beginning of the loop *update* [4, §14.12.2] portion.

3.5.4 Statement List

The statements of a Java method are contained in a **stmtList** object, which represents a single, flat list of statements. The **stmtList** class extends **zLinkedList** (see §4.4), and is largely responsible for maintaining the consistency of the statements in HLIR.

Through the design of the constructors and by overriding much of the functionality inherited from **zLinkedList**, the **stmtList** class is guaranteed to never represent a syntactically incorrect list of Java statements. Each construct (`e.g. if-then-else`) has an associated **stmtList** constructor, which enforces these restrictions. For example, it is not possible to create a statement list including an **ifStatement** unless the body of the *then-part* is also specified. Note, however, that the syntactic correctness of each statement used to construct the list is guaranteed

by the individual **zStatement** constructors.

The methods for transforming a **stmtList** (inserting or removing statements) enforce consistency and maintain the internal structure of HLIR. This is described in detail in §3.5.5.1 and §3.5.5.2. These methods also automatically maintain the symbol table information, as described in §3.8. Skeleton methods are provided, which are guaranteed to be called every time statements are added to or removed from the list. This allows users to maintain their own structures under transformations, if desired (see §4.2).

Methods are provided to obtain iterators over the list, using the `java.util.ListIterator` interface. As well, specialized iterators can be obtained, which iterate over only a specified set of statement types. That is, the specialized iterators will skip over statements which do not match the set of statement types specified when the iterator is created. The set is specified using a bitmask of integer statements types, allowing iterators over any combination of statement types to be created. Several of the methods of class **stmtList** are summarized in Table C.5 of Appendix C (page 119).

The **stmtList** class contains a constructor associated with each kind of Java construct, which allows for the creation of a new statement list containing that construct. For example, the constructor for an `if-then-else` takes as arguments an **ifStatement**, a list of statements representing the *then-part*, and a list of statements representing the *else-part*. This constructor will return a new **stmtList** containing the construct, with the **thenStatement**, **elseStatement**, and **endStatements** automatically inserted and linked. The resulting list of statements (and the links between them) is shown in Figure 3.7 on page 47.

Many constructs require that their body be a *block*. Examples of such constructs are synchronized blocks, try blocks, catch clauses, and finally clauses. The **stmtList** constructors for these types of language construct verify that the body provided contains a single block at the outermost level (the contents of this block are not important). We refer to a statement list containing such a body as a *block statement list*.

Other constructs require that the body be either a single statement, or a or a list of statements representing a single construct. We refer to a statement list containing such a body as a *one-construct statement list* or a *one-construct body*. The validity of these bodies is verified in the associated **stmtList** constructor. If a construct requires a one-construct body, and the body specified is not already enclosed in a *block*, it is automatically placed inside one. For example, if the provided *then-part* of an *if-then* construct is a single statement, it will be wrapped in a block (i.e. "{ }") as the *if-then* statement list is built. This simplifies much of the consistency enforcement if the construct is later modified.

The following sections describe the HLIR representation of each Java construct in detail. In the figures in these sections, an edge between statements represents that the statement at the base of the edge contains a reference to the statement at the head. Additionally, a **stmtList** enclosed in square brackets ("[" "]") is potentially an empty list.

3.5.4.1 Block Construct

A *block* construct is used to represent any (possibly empty) sequence of statements enclosed by cursive braces ("{" "}"). The structure of a `block` construct is shown in Figure 3.3. The **stmtList** constructor for a block requires a non-null **blockStatement**, as well as a non-null (but possible empty) **stmtList** to represent the body of the block.

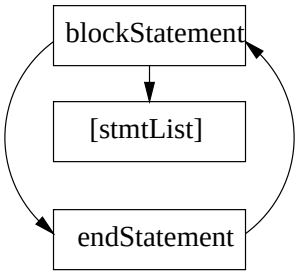


Figure 3.3: Structure of a block construct.

3.5.4.2 Labelled Construct

A label may be attached to any statement list which represents a one-construct body. The representation of a labelled statement is given in Figure 3.4. The **stmtList** constructor for a labelled construct requires a non-null **labelStatement**, as well as a non-null **stmtList** to represent the statement or construct to which the label applies. The constructor also verifies that the body provided is a one-construct statement list.

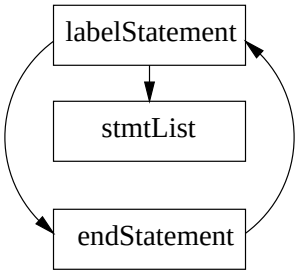


Figure 3.4: Structure of a labelled construct.

3.5.4.3 Try-Catch-Finally Construct

The representation of a `try-catch-finally` construct is shown in Figure 3.5. Note that this example contains only one *catch clause*; a *try* construct may contain zero or more *catch clauses* in general. The *finally clause* is optional, but in the absence of such a clause, the construct must contain at least one *catch clause*. The **stmtList** constructor for a *try* construct imposes these restraints. Additionally, this constructor requires the following:

- a non-null **tryStatement**.
- a non-null **stmtList** for the *try-block*, which must be a block statement list.
- a possibly-null list of **catchStatements**, one to represent the header of each *catch clause*.
- a possibly-null list of **stmtList** objects, one to represent the body of each *catch clause*. These **stmtList** objects must all be block statement lists.

- a possibly-null **finallyStatement**, to represent the header of the *finally clause*.
- a possibly-null **stmtList**, to represent the body of the *finally clause*. This **stmtList** must also be a block statement list.

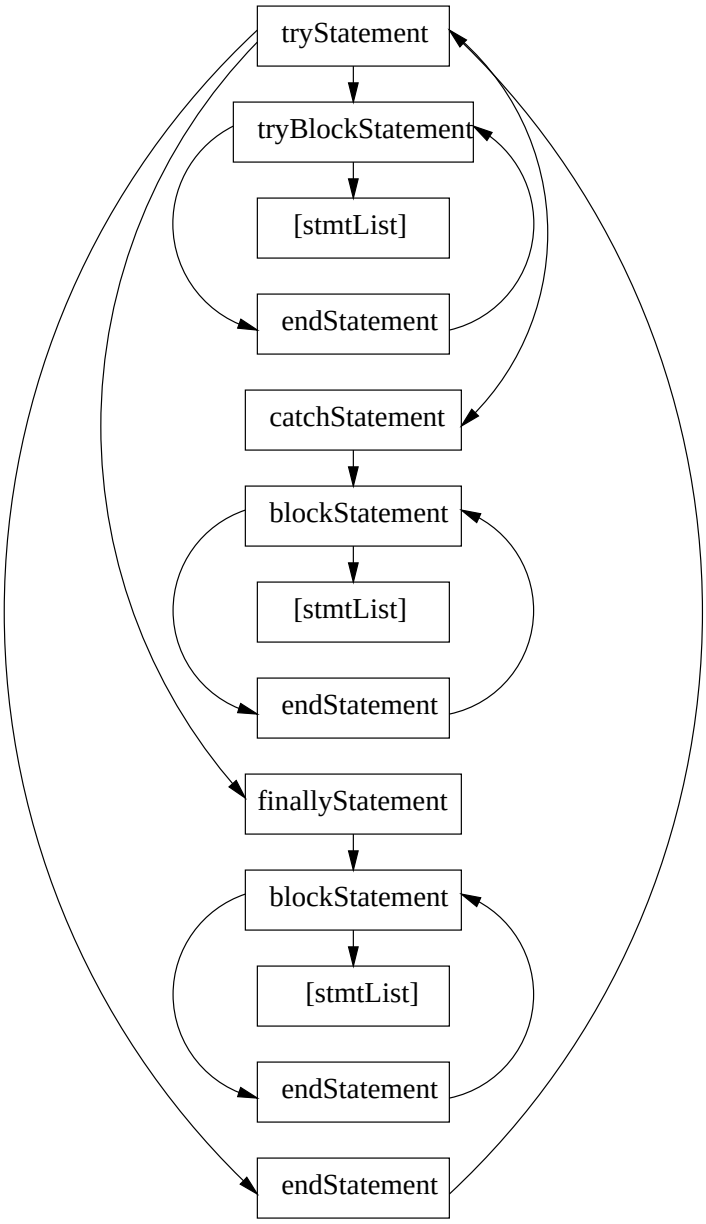


Figure 3.5: Structure of a try-catch-finally construct with one catch clause.

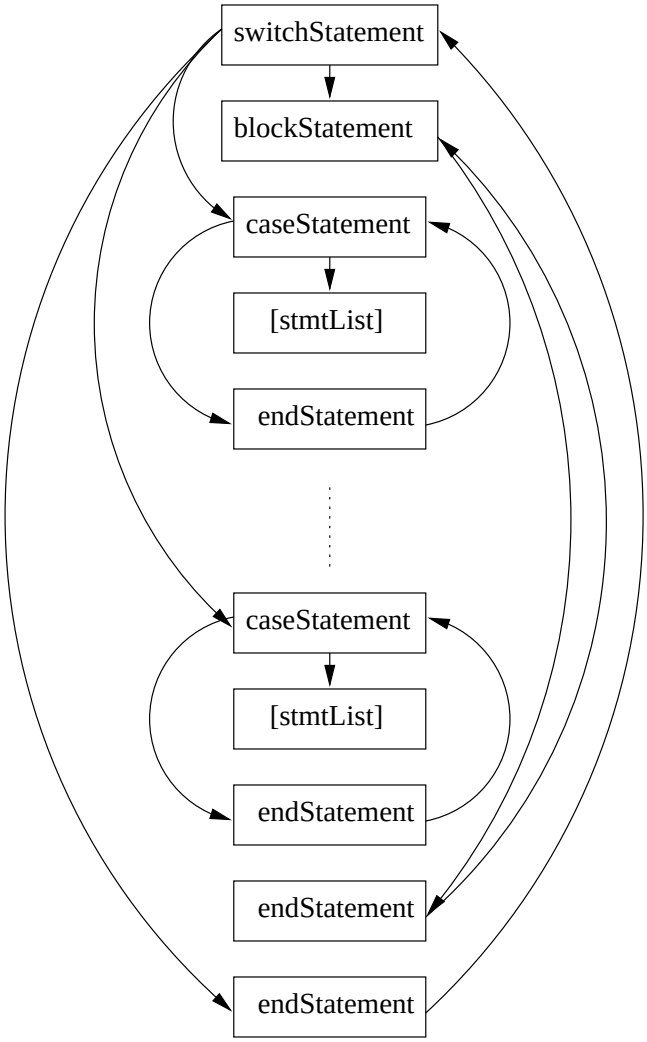


Figure 3.6: Structure of a switch construct.

3.5.4.4 Switch Construct

The representation of a `switch` construct is given in Figure 3.6. A `switch` construct may contain zero or more *case clauses*. The **stmtList** constructor for a `switch` construct requires a non-null **switchStatement**. The constructor also requires a possibly-null list of **caseStatements** and a possibly-null list of **stmtList** objects, to represent the headers and bodies of each *catch clause*.

3.5.4.5 If-Then-Else Construct

The representation of an `if-then-else` construct is given in Figure 3.7. An `if-then` construct is identical except that the else-part is not present. The **stmtList** constructor for and `if-then-else` requires a non-null **ifStatement**, a non-null **stmtList** to represent the *then-part*, and a possibly-null **stmtList** to represent the *else-part*. These statement lists must both be one-construct statement lists.

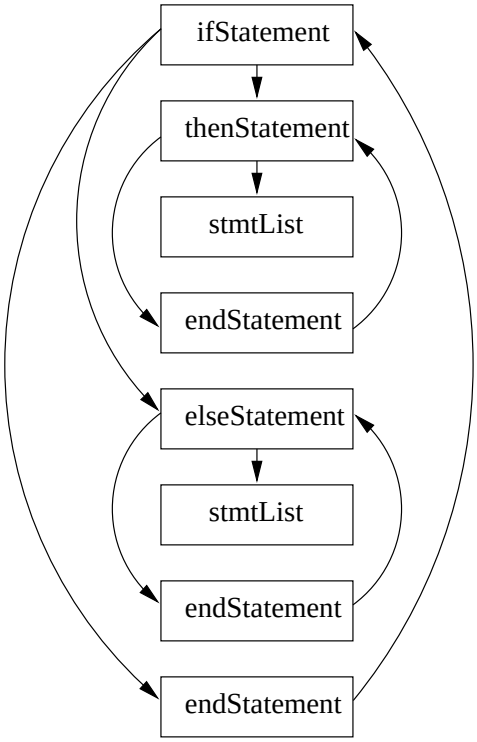


Figure 3.7: Structure of an if construct.

3.5.4.6 Synchronized Block

Figure 3.8 shows the representation of a `synchronized` block. The **stmtList** constructor for a `synchronized` block requires a non-null **synchronizedStatement**, and an non-null **stmtList** representing the body of the construct. This **stmtList** must be a block statement list.

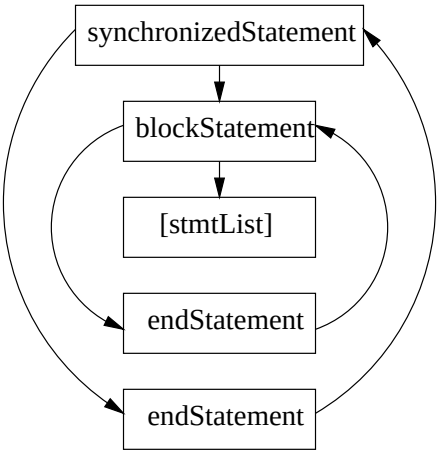


Figure 3.8: Structure of a synchronized construct.

3.5.4.7 Do and While Loops

Figure 3.9 shows the representation of a `do` loop. A `while` loop is identical except that the **doStatement** is replaced by a **whileStatement**. The **stmtList** constructors for do and while loops require a non-null **doStatement** or **whileStatement**, respectively. Both constructors also require a non-null **stmtList**, which must represent a one-construct body.

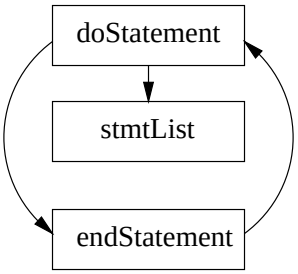


Figure 3.9: Structure of a do loop.

3.5.4.8 For Loop

The `for` loop construct is particularly difficult to flatten. Conceptually, a `for` statement contains a list of statements for the *for-init*, a list of statements for the *for-update*, and an expression (or a list of expressions) for the conditional. For most constructs, the statements from the AST representation can simply be inserted into

the flat statement list in order, with appropriate marker statements. Flattening a `for` construct involves taking the *for-init* block and inserting it into the flat list first, delimited by a **forInitStatement** and a corresponding **endStatement**. The *for-header* is then inserted, as a **forStatement**. The statements of the loop body then follow, and the construct is terminated by an **endStatement** linked to the **forStatement**. The statements of the *for-update*, delimited by a **forUpdateStatement** and an **endStatement**, are included as part of the loop body. The resulting list of statements is shown in Figure 3.10. Thus, the `for` construct begins with the **forInitStatement**, and ends with the **endStatement** corresponding to the **forStatement**.

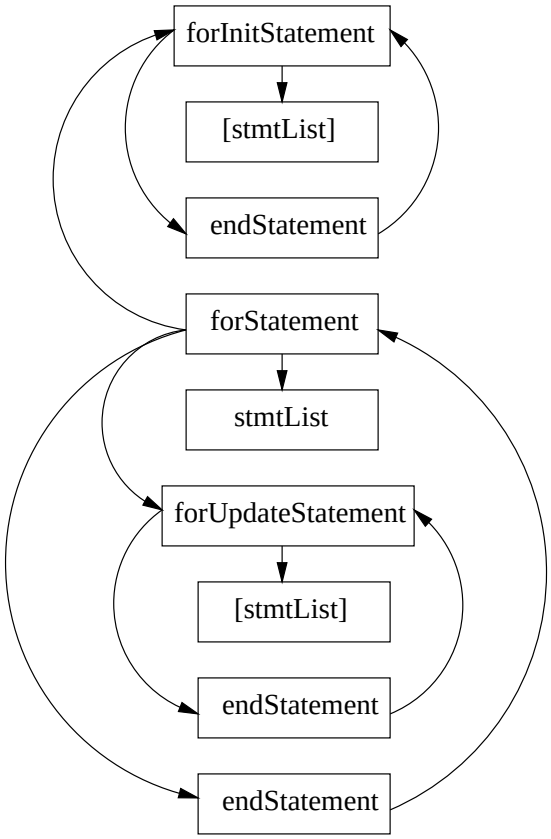


Figure 3.10: Structure of a for loop.

The **stmtList** constructor requires non-null **forStatement**, and a **stmtList** representing each of the loop body, *for-init*, and *for-update*. The latter two lists may be null, while the former must represent a one-construct body.

3.5.5 Statement List Consistency

The following two sections describe how consistency is maintained by the **stmtList** class when a statement list is modified.

3.5.5.1 Statement List Insertion

Insertion into a **stmtList** is restricted in such a manner as to guarantee both the syntactic consistency of the represented code, and the internal consistency of HLIR. The **stmtList** class does not allow the insertion of individual statements into a list; rather, only complete **stmtLists** may be inserted. In general, insertion of one fully-formed Java construct into another will result in a syntactically correct sequence of statements. Thus, statement lists containing sections of code can be constructed separately, then inserted into one another to create a complete method body. Similarly, correct statement lists can be created and then inserted into an existing method body. Note that a **stmtList** is *absorbed* when inserted into another **stmtList**—that is, all elements of the input list are removed, and added to the target list. Thus, at any time, a method body consists of a single **stmtList** object.

Figure 3.11 contains examples of two correct statement lists, one containing an `if-then-else` construct and the other a `while` loop. Each list can safely be inserted at several points into the other. For example, the `while` loop list could be inserted into the top list in these places: before or after element 3, before or after element 8; before element 0; after element 11.

Special cases, where the result of inserting one list into another is not guaranteed to be correct, are handled in the **stmtList** insertion methods. An example of such a case is the attempt to add statements after the closing "}" of a method body, or of certain constructs. For example, the following sequence is illegal:

```
try {
  // try body statements
}
illegalStatementHere();
catch (Exception e} {
  // catch clause statements
}
```

```
0:  ifStatement                        if (i < 3)
1:  thenStatement
2:  blockStatement                     {
3:  exprStatement                          ++i;
4:  endStatement    (end block)        }
5:  endStatement    (end then)
6:  elseStatement                      else
7:  blockStatement                     {
8:  exprStatement                          --i;
9:  endStatement    (end block)        }
10: endStatement    (end then)
11: endStatement    (end if)


12: whileStatement                     while (i < 10)
13: blockStatement                     {
14: exprStatement                          ++i;
15: endStatement    (end block)        }
16: endStatement    (end while)
```

Figure 3.11: Examples of two fully-formed statement lists, with equivalent Java source code.

Other illegal insertions involve inserting statements inside a construct in such a way that the internal structure of the construct is damaged. For example, in an `if-then-else` construct, the structure of HLIR requires that the **thenStatement** immediately follow the **ifStatement**. Inserting a statement between these two (i.e. between lines 0 and 1 in Figure 3.11) would break the structure of the construct. In fact, inserting statements between most construct headers and the first statement of their body leaves the IR in an inconsistent state, and is thus prohibited. A similar situation arises when trying to insert statements immediately after the end of a *block* which is the body of a construct, but before the **endStatement** which terminates the construct (i.e. after lines 4, 9, or 15 in Figure 3.11). Note that these examples refer to the consistency of the internal structure of HLIR, and not the syntactic consistency of the represented Java code.

Since `for` loops are flattened, and the *for-update* is placed inside the body of

the loop, a special case arises when attempting to to add to the end of a `for` loop body. Such insertions are detoured and inserted just before the *for-update* part, to guarantee the desired semantics.

Scoping of statement lists, and resolution of symbols to point to symbol table entries, is not performed until the **stmtList** becomes part of a method body. This happens if the list is set as a method body using `methodObject.setStmts(stmtList)`, or if it is inserted into the **stmtList** of an existing method body. The statements in a stand-alone **stmtList** (such as those in Figure 3.11) have null symbol table references, as no scope information is present. Similarly, symbols contained in the expressions within these statements are stand-alone symbol objects (rather than references directly into a symbol table). When a **stmtList** becomes part of a method body, its scoping information is determined. The scope information of the inserted list is merged with the scope information of the existing method body. This mechanism is described in detail in §3.8.1. As well, the types of all symbols and expressions in the new list are resolved, and expressions are altered to reference symbol table entries directly where needed.

3.5.5.2 Statement List Removal

Methods inherited from **zLinkedList** (§4.4) which allow for removal of elements from the list are overridden to only permit removal when the resulting **stmtList** will be correct.

Certain statements can always be removed, regardless of the context. These are:

- **breakStatement**
- **continueStatement**
- **emptyStatement**
- **exprStatement**
- **returnStatement**
- **throwStatement**

Attempting to remove the header of a compound construct results in the removal of the entire construct. For example, removing a **whileStatement** results in the removal of the **whileStatement**, its **endStatement**, and the block of statements in between (the body of the `while` loop). The permitted construct removals are summarized in Table 3.2. Statements which are sub-constructs, and can only be removed through specialized methods in the appropriate construct header, are summarized in Table 3.3.

Statement Type	Notes
blockStatement	
tryBlockStatement	
ifStatement	*then-clause*, and *else-clause* also removed, if present
labelStatement	
switchStatement	removes all *case clauses* as well
synchronizedStatement	
tryStatement	*catch clauses* and *finally clause* also removed, if present
doStatement	
whileStatement	
forStatement	removes *for-init* as well
forInitStatement	removes entire *for* construct

Table 3.2: Allowable construct removals

Statement	Associated Header Statement
elseStatement	**ifStatement**
caseStatement	**switchStatement**
catchStatement	**tryStatement**
finallyStatement	**tryStatement**

Table 3.3: Sub-constructs, which can be removed via special methods in the associated construct header

When the last statement in the body of a construct is removed, it is automatically replaced with an empty *block*. When the last statement in the body of a method is removed, it is automatically replaced with and **emptyStatement**.

It should be noted that certain statements can never be removed, because doing so would leave a construct in an invalid state. Attempting to remove such statements will always cause a run-time error. Statements which can never be removed are:

- **endStatement**
- **forUpdateStatement**
- **thenStatement**
- **implicitStatement**

3.6 Expressions

In this section we introduce the representation of expressions, and describe the individual expression classes in detail. We then discuss the details of type evaluation, followed by details of the representation of arrays. The section ends with a discussion of issues involved in the generating source code for expressions.

Expressions are represented in a tree format, with each expression potentially containing subexpressions. The base of the expression class hierarchy is the abstract class **zExpression**. This class provides the basic functionality common to all expression classes, including a reference to the **zStatement** containing the expression, and methods to access the type of the expression, return a list of all subexpressions, convert to Java source, and evaluate the type of the expression. The list of subexpressions is created by recursively traversing the tree of expressions referenced. Several of the methods of the **zExpression** class are detailed in Table C.6 of Appendix C (page 120).

3.6.1 Expression classes

In this section we detail specific **zExpression** classes, relating each to its Java source code equivalent.

The classes **StringLiteralExpression**, **booleanLiteralExpression**, **charLiteralExpression**, **doubleLiteralExpression**, **floatLiteralExpression**, **intLiteralExpression**, **longLiteralExpression**, and **nullLiteralExpression** represent literals in the source code. Table 3.4 lists these and other expressions which contain no subexpressions, with examples of their source code equivalents. Table 3.5 lists

the expressions which may contain subexpressions, and examples of their source code equivalents.

zExpression class	Java equivalent example
StringLiteralExpression	`"Hello"`
booleanLiteralExpression	`true, false`
charLiteralrExpression	`'a'`
intLiteralExpression	`42`
longLiteralExpression	`42l`
floatLiteralExpression	`2.0f`
doubleLiteralExpression	`2.0d`
classExpression	"`class`" in `Integer.class`
emptyExpression	no source equivalent
nullLiteralExpression	`null`
varExpression	`myVar`
superExpression	`super`
thisExpression	`this`

Table 3.4: Expressions which contain no subexpressions.

3.6.1.1 Array Access Expression

The Java 1.1 grammar considers an array access expression [4, §15.12] as an *array reference expression* followed by a single *index expression*. Thus, an expression such as `a[i][j]` would in fact be two array access expressions; the first has reference expression `a[i]` and index `j`; the second has reference expression `a` and index `i`. In the zJava HLIR, we represent an expression such as `a[i][j]` as a single **arrayAccess**, with reference expression `a` and a list of array index expressions, `(i, j)`. Thus, compiler passes see a multidimensional access of a single array as a single expression with a list of indices, rather than having to pick apart a cascade of array access expressions. The **arrayAccess** class includes a method to access the base expression, and a method to return the list of indices. Further details of the HLIR representation of arrays are given in §3.6.3.

zExpression class	Java equivalent example
arrayAccess	`a[5][6]`
binaryExpression	`a = b`
instanceOfExpression	`str instanceof String`
castExpression	`(String) j`
fieldAccess	`Foo.fd`
methodInvocation	`bar(a, b, 6)`
newArray	`new int[][][]`
newObject	`new String("hello")`
unaryExpression	(no source equivalent)
preUnaryExpression	`++a`
postUnaryExpression	`a--`
questionExpression	`1 == 0 ?  false :  true`
naryExpression	any list of expressions, such as parameters to method invocation, formal arguments, array access expressions

Table 3.5: Expressions which may contain subexpressions.

3.6.1.2 Binary Expression

Several different types of expressions from the Java grammar are grouped together in the HLIR class **binaryExpression**. Each **binaryExpression** consists of left and right subexpressions, and an operator, and the class includes methods to access these fields. A method is also included which determines (based on the operator) if the expression represents an assignment. Table 3.6 details the kinds of expressions represented by **binaryExpression**, with source code equivalents and references to appropriate sections of the JLS [4]. Note that the `instanceof` expression [4, §15.20.2] is represented by a separate HLIR class (§3.6.1.3).

3.6.1.3 Type Comparison Expression

The special binary expression `instanceof` [4, 15.20.2] (e.g. `str instanceof String`) is represented by the HLIR class **instanceOfExpression**. This class contains a reference to the *relational expression* (`str` in the example above), and the type of the *reference type* (`String` in the example), as well as methods to access each of these properties.

Expression type	**Operator**	**JLS Section**
Assignment	$=, *=, /=, \%=, +=, -=$ $<<=, >>=, >>>=, \&=, \mid=$	[4, §15.25]
Relational	$<, <=, >, >=$	[4, §15.19.1]
Conditional	&&, ||	[4, §15.22, §15.23]
Equality	$==, !=$	[4, §15.20]
Bitwise or Logical	&, ^, |	[4, §15.21]
Shift	$<<, >>, >>>$	[4, §15.18]
Additive	$+, -$	[4, §15.17]
String Concatenation	$+$	[4, §15.17]
Multiplicative	$*, /, \%$	[4, §15.16]

Table 3.6: Various binary expressions represented by **binaryExpression**.

3.6.1.4 Cast Expression

A cast expression [4, §15.15] is represented by the class **castExpression**. This class contains a reference to the *operand* (the castee), and to the type of the *cast operator* (the type to cast into), and a method to access each.

3.6.1.5 Field Access Expression

A field access expression, such as `System.out`, is represented by a **fieldAccess** object. The class **fieldAccess** contains a reference to the *base expression* (`System` in the example), and a reference to the *field expression* (`out` in the example) and methods to access each of these.

Note that cascaded field access expressions are represented by nested **fieldAccess** expressions. Thus, the expression `MyClass.description.colour` is represented by two **fieldAccess** objects—the first with base `MyClass.description` and field `colour`, and the second with base `MyClass` and field `description`.

Names are disambiguated [4, §6.5] during field access type evaluation. For example, when the expression `java.lang.System.out` is initially parsed, it appears to

be a cascade of field access expressions. However, `java.lang.System` is in fact the name of a class (including the package name, in this example), and there is in fact only one field access. Thus, we represent an expression like `java.lang.System.out` as a single **fieldAccess** object, with the expression `java.lang.System` as the base, and `out` as the field. Note that the base in this case is a single expression (containing the symbol named `java.lang.System`), rather than a field access expression. The details of name disambiguation are given in §3.6.2.6.

3.6.1.6 Variable Expression

The class **varExpression**, which contains either a single **varSymbol** or a single **classSymbol** (§3.7), is used to represent a *Name* [4, §6]. If the expression is contained in a statement which belongs to a method body (rather than a stand-alone **stmtList**), the symbol will be a reference directly into the appropriate symbol table. The class **varExpression** includes a method to access the symbol it contains.

3.6.1.7 Method Invocation Expression

We represent a method invocation expression [4, §15.11] with the **methodInvocation** class, containing the method name and a list of arguments. A qualified method invocation (eg, `str.concat(" World!")`, as opposed to the unqualified invocation `hello()`) is also considered a method invocation expression by the Java grammar [4, §19]. We choose to represent a qualified method invocation as a field access expression, with the qualifier (`str`, in the example) as the base, and the method invocation expression (`concat(" World!)`), in the example) as the field.

Note that explicit constructor invocations [4, §8.6.5] are also represented by **methodInvocation** objects, with the method name `super` or `this`.

3.6.1.8 Array Creation Expression

A array creation expression [4, §15.9] is represented by a **newArray** object, which consists of a reference to the type of the new array, a list of the *dimension expressions*,

and a method to access each of these. Note that Java1.1 includes two forms of array creation expression, only one of which is supported by HLIR (see §3.6.3).

3.6.1.9 Class Instance Creation Expression

A class instance creation expression [4, §15.8] is represented by the **newObject** class, which contains methods to access the type of the created object, and to access the list of arguments in the constructor invocation.

3.6.1.10 Unary Expressions

The abstract class **unaryExpression** contains functionality common to both prefix and postfix unary operators. The subclass **preUnaryExpression** represents prefix unary operators [4, §15.14], and the subclass **postUnaryExpression** represents postfix expressions [4, §15.13]. Methods are provided to access the *operator* and the *operand*.

3.6.1.11 `super` and `this` Expressions

The class **thisExpression** is used to represent uses of the keyword `this`, both in an explicit constructor invocation [4, §8.6.5] and in a *Primary* expression [4, §15.7.2]. Qualified `this` expressions (e.g. `OuterClass.this`) are an inner class issue, and will cause an exception during type evaluation.

The **superExpression** class represents uses of the keyword `super`, both in an explicit constructor invocation [4, §8.6.5] and in as a qualifier in a method invocation [4, §15.10.2]. Qualified `super` expressions are an inner classes issue, and will cause an exception during type evaluation.

3.6.1.12 The Conditional `?  :` operator

The Conditional `?  :` operator [4, §15.24]) uses the boolean value of a *conditional expression* to determine which of its other two subexpressions (a *true expression* and a *false expression*) to evaluate. An expression using the conditional `?  :` operator is

represented by a **questionExpression** object in HLIR. The class **questionExpression** provides methods to access each of the three subexpressions.

3.6.2 Type Evaluation

Most expressions are by default created with *invalid* type, as their type can not be determined until HLIR is populated and symbol table lookups are possible. The `evaluateType` method, overridden in each subclass of **zExpression**, is used to resolve the type of each during HLIR generation, and every time new statements are added to an existing method body. Where necessary, this method will query the appropriate symbol table (that is, the symbol table referenced by the statement containing the expression) to resolve the type of the symbols in the expression. The type returned is the static (compile-time) type, as shown in the example below:

```
class A {
  int x;
}

class B extends A {
  String x;
  void foo() {
    A a = new B();
    a.x = 3;
    ((B)a).x = "hi";
  }
}
```

In this example, the type of the expression `a.x` is `int`. The type of the local variable `a` evaluates as a reference to `A`, even though the object it references at run-time is of type `B`. This is the correct type evaluation, following the specification in [4, §15.10.1].

Table 3.7 lists **zExpression** derived types which contain no subexpressions, and the result of type evaluation for literal expressions. In the following sections, we first discuss the type evaluation of qualified expressions, then describe in detail the type evaluation of non-literal expressions.

zExpression class	Type signature
StringLiteralExpression	reference to java.lang.String
booleanLiteralExpression	boolean
charLiteralrExpression	char
intLiteralExpression	int
longLiteralExpression	long
floatLiteralExpression	float
doubleLiteralExpression	double
classExpression	reference to java.lang.Class
emptyExpression	<invalid type>
nullLiteralExpression	<null type>
varExpression	See §3.6.2.7
superExpression	See §3.6.2.12
thisExpression	See §3.6.2.12

Table 3.7: Expressions which contain no subexpressions, with types.

3.6.2.1 Qualified Expressions

The type of the field in a qualified expression cannot be resolved without first determining the type of the base expression (the qualifier). That is, given the expression `str.concat("a")`, the type of `str` (which might be a type name, or a local variable of reference type) must be known, before the type of the method `concat("a")` can be established. In certain cases, it is possible for the qualifier to be of an array type (Java arrays are implicitly considered to be objects), although the HLIR does not support type evaluation of expressions with an array as the qualifier (see §3.6.3). Qualifiers of a primitive type are illegal, and will result in an exception during type evaluation.

The `evaluateType` methods in the various expression classes can accept the type of the qualifier as a parameter. In many cases this has no meaning (eg., a pre-increment expression with a qualifier, such as `myRef.++a`, is illegal). If a non-null qualifier is passed to `evaluateType` in a situation where it has no meaning, a run-time exception is raised.

The `evaluateType` method is designed to be invoked on complete expressions (i.e. the top level of an expression tree), rather than on individual subexpressions. Thus, in the example above, rather than determining the type of `str`, and then passing that

as the qualifier when evaluating the type of `concat("a")`, one should instead invoke `evaluateType` on the entire field access expression, `str.concat("a")`. *Note that, in general, it should not be necessary to explicitly invoke type evaluation. The types of expressions and symbols are resolved automatically, even when new statements are added to an existing method body.*

3.6.2.2 arrayAccess

An array access expression evaluates to the type of its *array reference expression*. The number of array dimensions is taken as the declared dimensions of the symbol in the array reference expression, minus the number of index expressions in the **arrayAccess** expression. Thus, in the code example given below, the expression `a[3]` type-evaluates as a two-dimensional array (`int[][]`).

```
int[][][] a = new int[5][3][2];
a[3] = { {2, 4}, {6, 8}, {10, 12} };
```

Note that the type of each index expression is also evaluated during type evaluation of the **arrayAccess**, even though this is not necessary to determine the type of the array.

3.6.2.3 binaryExpression

The set of expressions represented by **binaryExpression** is detailed in Table 3.6 on page 57. In many cases, it is not necessary to evaluate the type of both subexpressions of a **binaryExpression** in order to determine the type of the overall expression. Nonetheless, the type of both operands is always evaluated, such that the type information in the representation is complete.

In the case of a binary expression representing an assignment, the type of the **binaryExpression** is the type of the left-hand side. If the operator is a relational operator, a conditional-and or conditional-or operator, or an equality operator, the type evaluates to `boolean`. If the operator is a bitwise or logical operator, the type evaluates to `boolean` if both the left and right operands evaluate to `boolean`. Otherwise, the type evaluates to the result of *binary numeric promotion* (see §3.6.2.14)

on the two operands. If the operator is a shift operator, the expression evaluates to the result of *unary numeric promotion* (see §3.6.2.14) on the left-hand operand. For additive operators, the type is the result of binary numeric promotion on the left and right operands. However, if the operator is +, and either operand is a reference to type `java.lang.String`, then the binary expression evaluates as a reference to `java.lang.String` (and the operator is considered to be the string concatenation operator). Finally, if the operator is multiplicative, the expression evaluates to the result of binary numeric promotion on the operands.

3.6.2.4 instanceOfExpression

An **instanceOfExpression** always evaluates to the primitive type `boolean`. The type of the relational expression is also evaluated.

3.6.2.5 castExpression

The type of a **castExpression** always evaluates to the type of the *cast operator* [4, §15.15]. The type of the operand is also evaluated.

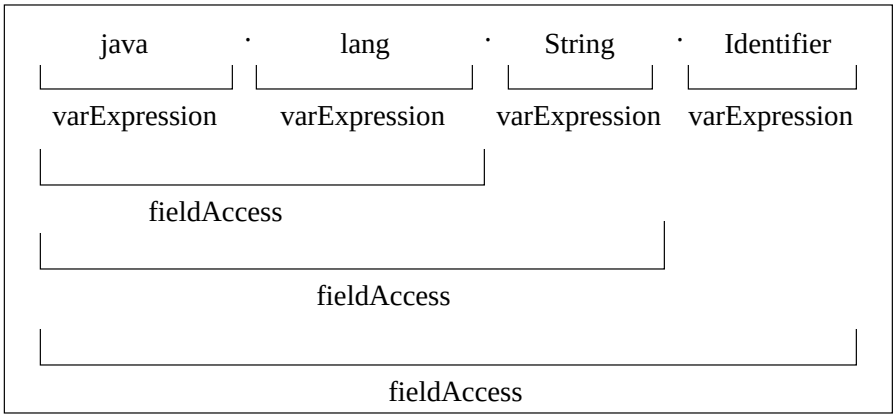
3.6.2.6 fieldAccess

Type-evaluation of a field access expression [4, §15.10] first evaluates the type of the *base*. The type of the **fieldAccess** is then the result of type-evaluating the *field* expression, given the type of the base as a qualifier.

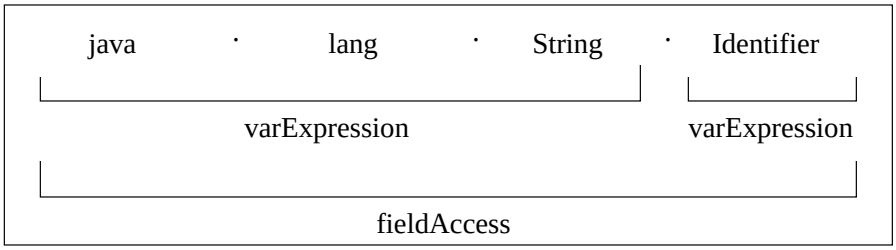
Names are disambiguated [4, §6.5] during field access type evaluation. At parse time, it is not always possible to determine if a name represents a variable, a type name, or a package name. Names which include a *PackageName* are broken into one or more field access expressions by the front end. These are collapsed into a single **varExpression** during the type evaluation stage. Figure 3.12 gives an example.

The algorithm for name disambiguation proceeds as follows:

1. First, the base is treated as a symbol name. If the base can be found in the symbol table, it is the name of a local variable, a field, or an inherited field. If



a) Ambiguous name, originally split into separate varExpressions



b) After disambiguation; "java.lang" has been determined to be a package name, and "String" a Type in that package.

Figure 3.12: Example of Name disambiguation.

the symbol table query fails, the algorithm proceeds to step 2. Otherwise, the type of this symbol is determined. The **ClassShell** defining this type is located (or, if this is a primitive type, an error is raised), and its symbol table is queried for a symbol matching the field part of the **fieldAccess**. If this query fails, an error is raised. In the example in Figure 3.12, this step would involve querying the symbol table of the current scope for a symbol named "`java`".

2. If the base is not a visible symbol, it is treated as a type name, and an attempt is made to locate the **ClassShell** defining this type. If the class cannot be located (§3.2.1), the algorithm proceeds to step 3. Otherwise, the symbol table of this class is queried as above. If this query fails, an error is raised. In Figure 3.12, this step would involve an attempt to locate a class named "`java`".

3. If the base is not a type name, it is treated as a package name. An attempt is

made to locate an accessible class matching this name. This involves searching the directories in the list specified by the environment variable `CLASSPATH`. If any of these directories contains a subdirectory with a name matching the base of the **fieldAccess**, the base is treated as a package name. In Figure 3.12, this step would involve an attempt to locate a package named "`java`". If this search succeeds, the package is searched for a either subpackage or a class, whose name matches the field part of the **fieldAccess**. In Figure 3.12, this corresponds to searching the package "`java`" for a subpackage or class named "`lang`". If neither a subpackage nor a class can be found, an error is raised. Otherwise, the **fieldAccess** is collapsed—that is, the base and the field are combined as a single symbol name, joined by a ".".

In the example in Figure 3.12, the type evaluation of the field access `java.lang.String.Identifier`, and associated disambiguation, would proceed as follows:

- Invoke type evaluation on the base, which is a field access (`java.lang.String`).
  - Invoke type evaluation on the base of this field access, which is another field access (`java.lang`).
    - Invoke type evaluation on the base of this field access (`java`). This will determine that `java` is a package name.
    - Search this package for a subpackage or class named `lang`. This will locate a subpackage named `java.lang`.
    - Collapse this field access into a single expression, `java.lang`
  - Invoke type evaluation on the field (`String`), with the package `java.lang` as a qualifier. This will locate the class `java.lang.String`.
  - Collapse this field access into a single expression, `java.lang.String`.
- Invoke type evaluation on the base (`Identifier`), with the type `java.lang.String` as a qualifier. This will query the symbol table of class

`java.lang.String`, searching for a symbol named `Identifier`. If this symbol is found, its type will be returned (as the type of the overall expression).

3.6.2.7 varExpression

If no qualifier is specified, type evaluation of a variable expression follows these steps:

- Search for a matching **varSymbol** in the current scope. This will identify local variables, method parameters, class fields, and accessible superclass fields. The resulting type is the type of the matching symbol.
- Otherwise, search for a matching **classSymbol** (an accessible *Type*). The resulting type is a reference to the *TypeName*.
- Otherwise, search for an accessible package whose name matches the symbol. The resulting type is a reference to the package name, and is flagged as representing a *PackageName*.

If a qualifier is specified, and it has been flagged as representing a PackageName, the following sequence applies:

- Look for a matching **classSymbol** as a member of the qualifying package. The resulting type is a reference to the matching *TypeName*, with the qualifying package name prepended.
- Otherwise, look for a matching sub-package in the qualifying package. The resulting type is a reference to the package name, with the qualifying package name prepended.

If a qualifier is specified, and the qualifier does not represent a *PackageName*, the symbol table of the qualifying *Type* is searched for a matching **varSymbol**. The resulting type is the type of the matching class field.

In each case, when a matching symbol is found (either a **varSymbol** or a **classSymbol**), the **varExpression** is updated to reference the symbol table entry directly. If the symbol represented part of a *PackageName*, no matching symbol

table entry will be found until the **fieldAccess** containing this expression completes its own type evaluation and disambiguates the name (see §3.6.2.6).

The process of type evaluation on a variable expression may result in external classes being loaded from `.class` files, either when searching superclasses for a **varSymbol** matching the symbol, or when trying to find a **classSymbol** representing a *Type* which matches the symbol.

3.6.2.8 methodInvocation

Method invocation expressions are type-evaluated by searching the symbol table of the declaring class. If the expression is unqualified, the symbol table of the current class is examined (that is, the class containing the statement containing this expression). If the method invocation is qualified, then the class examined is that to which the qualifying type is a reference.

Method invocations are matched to method declarations based on the order, number, and type of arguments. Often, a method invocation will not directly match any declaration, and *method invocation conversion* [4, §5.3] is required to promote the arguments until a match is found. HLIR does not support method invocation conversion, meaning that a match between invocation and declaration is found only when the name, number of arguments, and types of arguments in the invocation match that of a declared method.

If a matching **methodSymbol** is found, the resulting type is the return type of that method, and the **methodInvocation** is modified to reference this symbol table entry directly. Otherwise, the resulting type is the invalid type, and the **methodInvocation** references a stand-alone **methodSymbol** (one which is not contained by any symbol table).

Explicit constructor invocations [4, §8.6.5] are handled by converting the method name to `<init>` before performing the symbol table lookup.

The type of each argument is also evaluated during the type evaluation of a method invocation expression.

3.6.2.9 newArray

A array creation expression [4, §15.9] evaluates to the type of the created array. If any *dimension expressions* are present in the **newArray** expression, their type is also evaluated.

3.6.2.10 newObject

A class instance creation expression [4, §15.8] evaluates to the type of the created object. If the constructor invocation in the **newObject** expression includes any arguments, their type is also evaluated.

3.6.2.11 preUnaryExpression and postUnaryExpression

Type evaluation is always performed by the base class **unaryExpression**. When the operator is a prefix or postfix increment or decrement ($++, --$), the type evaluates to the type of the operand. Prefix expressions containing the logical complement operator (!) evaluate to `boolean` type. Expressions containing any of the remaining prefix operators ($+, -, \tilde{}$) evaluate to the result of unary numeric promotion on the operand.

3.6.2.12 superExpression and thisExpression

The type of a **thisExpression** evaluates to the type of the enclosing class. Qualified `this` expressions are an inner class issue, and will cause an exception.

The type of a **superExpression** evaluates as the type of the superclass of the enclosing class. Qualified `super` expressions are an inner class issue, and will cause an exception.

3.6.2.13 questionExpression

Evaluating the type of a **questionExpression** (the Conditional `?  :` operator [4, §15.24]) is non-trivial. If both the second and third operands have the same type, the

expression evaluates to this type. If the second and third operands are of numeric types or differing reference types, several cases occur.

When one of the second or third operands is reference type, and the other is of the null type, the expression evaluates to that reference type. When the second and third expressions are of different reference types, it must be possible to convert one type to the other by *assignment conversion* (see §3.6.2.14). The zJava HLIR does not support assignment conversions, and thus cannot type-evaluate this form of a **questionExpression**.

In every case where one or both of the operands is of reference type, it is important to note that the resulting type of the expression is a compile-time type, based on the declared types of the operands. This may differ from their run-time types, as shown in the following example:

```
Object a = new String();
Object b = new String();
Object c = (true) ? a : b;
```

This expression will evaluate as a reference to `java.lang.Object`, while the run-time type will be `java.lang.String`.

If the second and third operands are both of numeric type, several possibilites exist [4, §15.25], all of which are handled by HLIR.

3.6.2.14 Type Conversion and Promotion

Type conversions and promotions in Java are quite complex [4, §5]). For the sake of static type evaluation, zJava includes methods for determining the result of unary and binary numeric promotion [4, §5.6.1, §5.6.2]. All other type conversions and promotions are not supported. This means, for example, that the type of a conditional `? :` expression, where the second and third operands are of different reference types, cannot be evaluated. As well, it will not always be possible to match a method invocation expression to a method declaration, meaning that it will sometimes not be possible to find the **methodObject** representing the method invoked by a **methodInvocation** expression.

3.6.2.15 Nested Classes

The zJava HLIR does not include support for inner classes [14]. However, where applicable, the type evaluation methods are aware of expressions specific to nested classes, and the framework is in place for extending HLIR to support these. Examples include *outer this* expressions, anonymous class instantiation expressions (qualified object and array creation expressions), and qualified explicit constructor invocations.

3.6.3 Arrays

Java arrays are considered to be objects that implement the interfaces `java.io.Serializable` and `Cloneable`, extend the class `java.lang.Object`, and include an implicit field `public final int length` and an implicit method `public Object clone()` (which does not override `java.lang.Object.clone()`). The type of an array (as returned by `java.lang.Class.getName()`, or `javaType.signature()`) consists of a "[" for each dimension, followed by the type signature of its elements. For example, a two dimensional array of `String` objects has type `[[Ljava.lang.String`. The zJava HLIR does not represent the fact that arrays are objects, nor the fact that they have implicit `length` and `clone` members. As a result, attempting to type-evaluate an expression that represents a field access on an array will result in an exception. Examples of these and other expressions involving array syntax are given in Figure 3.13.

The sub-arrays in one dimension of a multidimensional Java array are not all required to be of the same length (see [4, §10.9.2, §15.9.1]). Such a non-uniform array cannot be generated with a single array creation statement, but Figure 3.13 gives an example of how one can be created. The HLIR representation for types includes the type of, and number of dimensions in, an array. The length of each array dimension is not considered as part of the type—thus, the representation of non-uniform arrays does not present any difficulties.

Arrays can be initialized when they are declared, as seen in the two examples in Figure 3.13. The zJava frontend does not accept the second form of array initial-

```
// Unusual declaration syntax:
String[][] as[], bs[][], cs;
// The above declaration can be rewritten as
//   the three declarations below.
String[][][] as;
String[][][][] bs;
String[][] cs;

// Arrays are objects:
int [][]a;
a = new int[3][5];
System.out.println(a.length);
System.out.println(a.getClass.getName());

// Array initialization:
// The following two statements are equivalent
a = { {1, 2}, {3, 4} };
a = new int[][] { {1, 2}, {3, 4} };

// A non-uniform multidimensional array (from JLS 15.9.1):
float triang[][] = new float[100][];
for (int i = 0; i < triang.length; i++)
  triang[i] = new float[i+1];
```

Figure 3.13: Examples of array declarations, array initializers, and field accesses on an array.

ization, which was introduced in Java1.1 [13]. When an array declaration with an initializer is encountered, an implicit statement is inserted to represent the execution of the initializer (in the same manner as any other local variable declaration).

When a new array object is created, each of its elements is initialized to its standard default value [4, 4.5.4]. As this initialization is performed by the virtual machine, and is not explicit in the code, the zJava HLIR does not represent the implicit initialization of array values.

3.6.4 Source Code Regeneration

When converting expressions to source code, care must be taken to preserve the original semantics of the program. Hence, by default, the following expressions are

wrapped in parentheses when output as source code:

- **binaryExpression**
- **castExpression**
- **fieldAccess**
- **instanceOfExpression**
- **methodInvocation**
- **naryExpression**
- **questionExpression**
- **superExpression**

However, produce syntactically correct code, parentheses are eliminated in the following three cases:

1. the expression is the left-hand side of a **binaryExpression**
2. the expression is the *field* part of a **fieldAccess**
3. the expression is the top-level expression in an **exprStatement** or an **implicitStatement**

3.7 Symbols

The classes of the **zSymbol** hierarchy are used to represent symbols, which exist both in expressions and in the symbol table. The base class, **zSymbol**, contains properties shared by all types of symbols: name, type, and modifier (`public`, `private`, etc). Methods are provided to access and change these fields. Classes are derived from **zSymbol** to represent specific symbol types. Each of the derived types of symbols resides in its own namespace in the symbol table; for example, it is possible to represent a local variable and a method with the same name in the same scope.

The symbol for a single class is contained in a **classSymbol** object, which includes the superclass name, a list containing the names of implemented interfaces and superinterfaces, as well as flags to indicate if the object is a class or an interface. A **classSymbol** should only be seen in a **varExpression** or in the symbol table for a **compilationUnit**, as nested classes are not currently supported.

The **methodSymbol** subclass adds a list of **javaType** objects representing types of the formal arguments of the method. It also provides a method for calculating the signature of a method. A method signature is constructed from the method name and the types of its formal arguments. The return type is not included, as it is not needed to uniquely identify a method within its class.

The **varSymbol** subclass adds a **zExpression** object to represent the (optional) initializer of the symbol. It also includes a constructor which allows a **varSymbol** to be created from the information in a `java.lang.reflect.Field` object.

The **labelSymbol** subclass is used to represent the labels on labelled statements and in *break* and *continue* statements with target labels.

3.8 Symbol Table

The zJava HLIR uses the **symboltable** class to store symbol information and implement symbol lookups. The scoping information is established in a pass over a method body when the method is created, and automatically updated whenever symbol information changes. Examples of transformations which alter symbol table information are modifications to a statement list, addition or removal of class members, etc. Each **symboltable** represents a specific scope, and includes the symbols declared at the beginning of that scope, as well as a link to the parent (enclosing) scope. Symbol tables are thus arranged in trees, where the root of each tree is the symbol table for a single class. The set of symbols available in a particular scope include all symbols in its **symboltable**, and in all ancestors of that table. Symbols exist for variables, labels, methods and classes, with a separate namespace for each type.

A new scope is created by the following situations:

- Beginning of a class
- Beginning of a method
- Beginning of a statement block
- Local variable declaration within a block
- Beginning of a *for* construct
- Beginning of a *catch* construct
- Beginning of a *labelled* construct

The scope created for a local variable extends to the close of the `block` in which it was declared[3]. The scope created for a *catch clause* extends from the **catchStatement** to its **endStatement**, and represents the fact that a **catchStatement** declares a *catch exception* variable. The scope generated for a *labelled* construct extends from the **labelStatement** to its **endStatement**, and represents the fact that the label is visible within the labelled construct. The scope created for a `for` construct begins at the **forInitStatement** and extends to the end of the construct (the **endStatement** linked to the **forStatement**). This is necessary to represent the fact that a *for-init* part can declare new local variables. It is important to note, however, that while the statements of the *for-update* reside in the loop body, they are not necessarily in the same scope. This is demonstrated in Figure 3.14, where the for-update is moved to the end of the loop body (line 7 on the right side), but still resides in the scope of the for-header (line 1 on the left side of the figure).

An example of the symbol table representation is given in Figure 3.15, which demonstrates the scoping of an `if` construct.

The **symboltable** class includes methods to insert and remove symbols, to obtain a list of all symbols in a table (in the order in which they were inserted), to access the parent of a symbol table, and to search for symbols. Symbol insertion checks

[3]Note that local variables declared in a *case clause* exist until the end of the `switch` construct, rather than going out of scope at the end of the *case clause*.

```
0:  int i;
1:  for (i = 6; i < 10; ++i)
2:  {
3:    int j = i;
4:    write(j);
5:  }
```

```
0:  int i;
1:  i = 6;
2:  for (i < 10)
3:  {
4:    int j;
5:    j = i;
6:    write(j);
7:    ++i;
8:  }
```

Figure 3.14: A `for` loop, and the equivalent "flat" representation.

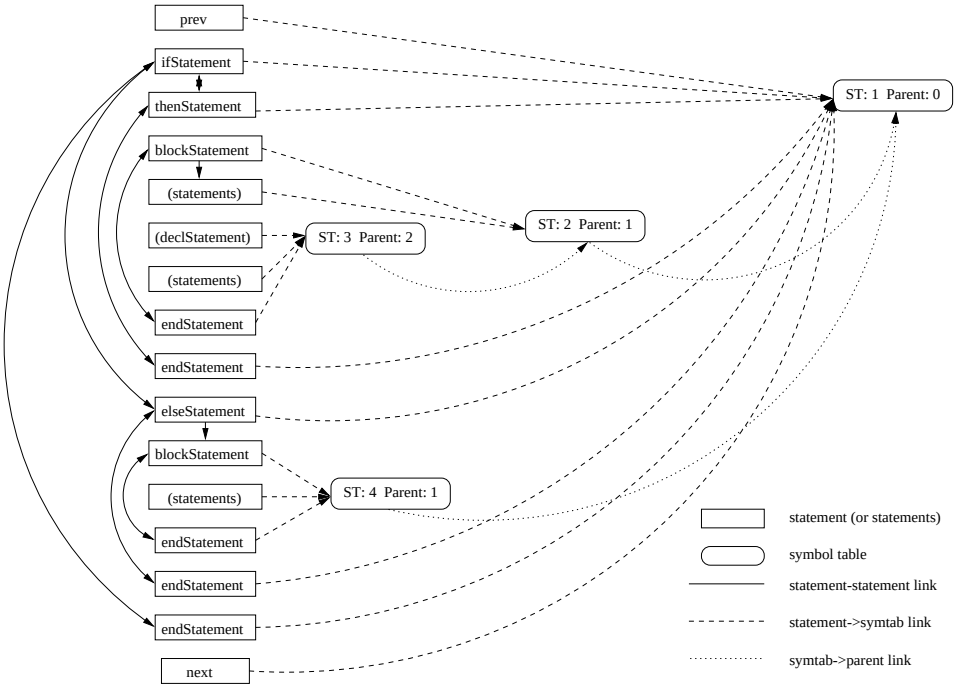


Figure 3.15: Scoping/symbol table representation for an `if` construct.

for a conflicting symbol name (in the same namespace), and raises an exception if a naming conflict occurs. The visible scope can be searched for variables, methods, classes, or labels, either by name or by specifying a **zSymbol** object to search for. Searching for a method symbol is successful only when the method signature of the specified symbol (from a *method invocation*, for example) exactly matches that of a symbol in the table. No attempt is made to promote arguments in order to match method parameters (see §3.6.2.8 for details).

When a compiler pass is iterating over the statements of a method body, it is often important to know when new variables are declared. The symbol table is the only source of this information, since declaration statements are removed during HLIR population, and variable declarations are turned into symbol table entries. The `genSet` method provides this functionality, determining what new symbols were declared between the specified **symboltable** and that on which it is invoked. Similarly, the `killSet` method will return the list of symbols which went out of scope.

3.8.1 Incremental Symbol Table Updates

When statements are added to an existing method body, the symbol and scope information of the new list must be merged into the existing symbol table tree. Figure 3.16 gives an example of the automatic incremental update of symbol table information, demonstrating the insertion of a statement list with a single statement (line 2 in Figure 3.16(b)), which introduces a new scope (symbol table 3 in Figure 3.16(b)).

In general, the algorithm for updating the symbol and scope information is as follows:

- *Determine insertion scope.* Determine *scope_insert*—the scope (in the existing method body) into which the new list will be inserted. In Figure 3.16(a), this is symbol table 1. The determination of *scope_insert* depends on the scope of the statement before the insertion point (*scope_before*), and the scope of the statement after the insertion point (*scope_after*). If these are the same scope (as is the case in Figure 3.16(a)), then this is used as *scope_insert*. If *scope_after* is nested within *scope_before* (i.e., is a descendant of it in the symbol table tree), then *scope_before* is the insertion scope. Finally, if *scope_before* is nested within *scope_after*, this means that the insertion is taking place just after the close of *scope_before*. In this case, *scope_after* is insertion scope.

- *Scope the new list.* Traverse the new list as described in §3.8, to determine its scope information, create and populate the associated symbol tables, and link each statement to the appropriate symbol table. In the simplest case, where the

new list does not introduce any new scopes, all of its statements will reference the symbol table of *scope_insert*. If the new list does introduce new symbol tables, these will be added to the symbol table tree under *scope_insert*.

- *Propagate new scopes.* If the new list introduced a new scope (*scope_new*), which is still open at the end of the new list, then the insertion of the new list changes the scope of statements after the insertion point. This new scope must be propagated forwards in the original method body. In Figure 3.16(b), the new list introduces a new scope (symbol table 3), which is still open at the end of the inserted list. The new scope must be propagated from the end of the new list until the close of *scope_insert* (note that in Figure 3.16, the close of *scope_insert* is somewhere after line 8). The statements in this range are part of the existing method body. Any statements in this range which link to *scope_insert* must be updated such that they belong to *scope_new* instead (e.g. lines 3 and 8 in Figure 3.16(b)). Further, any symbol tables in this range whose parent is *scope_insert*, must be altered to have *scope_new* as the parent. In Figure 3.16(b), this is reflected by the fact that symbol table 2 is altered to have symbol table 3 as its parent.

- *Type-evaluate new statements.* Symbols within expressions in HLIR statements are direct references to the appropriate symbol table entry. Expressions within the statements of the new list will by default have *invalid* type, and symbols in these expressions not reference symbol table entries. After the statements have been added, and the symbol table information has been updated, type evaluation (§3.6.2) is invoked on the expressions of the new list. This will resolve the type of each symbol and expression in these statements, and will alter symbol references to be refer directly to symbol table entries. Additionally, The symbol table of *scope_new* may contain a symbol which hides a declaration in *scope_insert*[4]. If so, any existing method body statements, which have been altered so as to belong to *scope_new*, must be examined. In the expressions

[4]Note that such a declaration would be semantically incorrect. It is illegal to declare the same symbol more than once within a block. A hiding declaration could exist inside a block within the

contained in these statements, any references to the now-hidden symbol must be altered to reference the symbol in the symbol table of *scope_new*. This is achieved by invoking type evaluation on these statements as well.

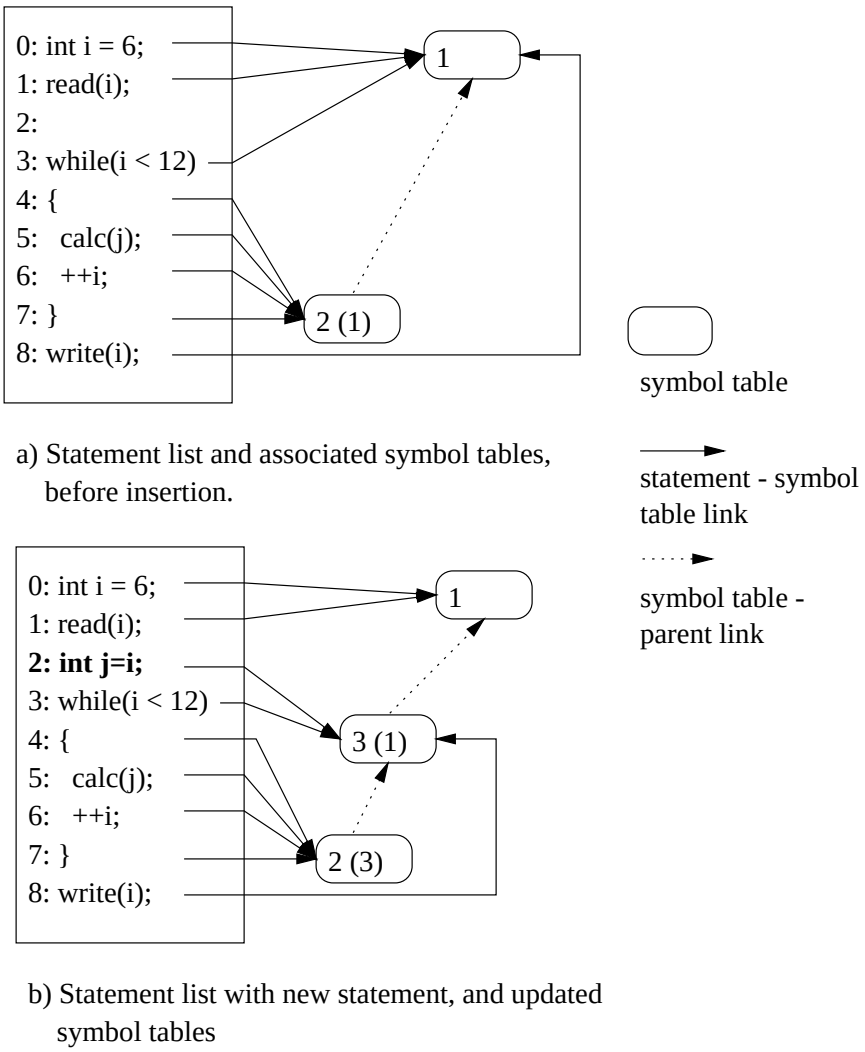


Figure 3.16: Automatic symbol table updates on statement insertion.

new list, in which case there is no need for special treatment. The syntactic consistency enforcement mechanisms guarantee that this block (and thus, the scope containing the hiding declaration) closes before the end of the new list.

3.8.2 Superclass Symbols

The class **SuperSymbolTable** is derived from **symboltable** as a special case to deal with superclass symbols. This class is extended to include the ability to find and load the superclass from bytecode if necessary, while presenting the same interface as **symboltable**. This hides the fact that superclass source code may not be part of the **Program** being analysed. The symbol table for each class has as its parent a **SuperSymbolTable** representing its superclass. In this way, members of superclasses are also examined when a symbol table lookup is performed. In Figure 3.17, the use of symbol `a` in method `b()` is resolved to the instance variable `a`, inherited from class `A`. The symbol lookup is performed by simply querying the symbol table for the scope containing the print statement. The query method traverses the tree up to the **SuperSymbolTable** labelled "*", which locates the **ClassShell** representing class `A` (loading it from bytecode if necessary). The **SuperSymbolTable** then queries the symboltable of class `A`, looking for a `public|protected` instance variable named `a`. If this variable was not found, the query would proceed to the superclass of `A`, which in this case is `java.lang.Object`. The resolution of superclass symbols is transparent; that is, any symbol table lookup automatically finds available symbols, even those which are inherited.

3.9 Access Modifiers

The **zModifier** class hierarchy is used to represent access modifiers of classes, methods, and fields in the program. The base class includes the flags common to all three: `public`, `protected`, `private`, `static` and `final`. The **classModifier** class adds the `abstract` modifier; the **methodModifier** class adds `abstract`, `native`, and `synchronized`; and the **fieldModifier** class adds `transient` and `volatile`. Each class provides methods to check and set each modifier. Class **zModifier** includes one other flag, `package`, which is used to represent the default access permission when no modifier is specified in the source code.

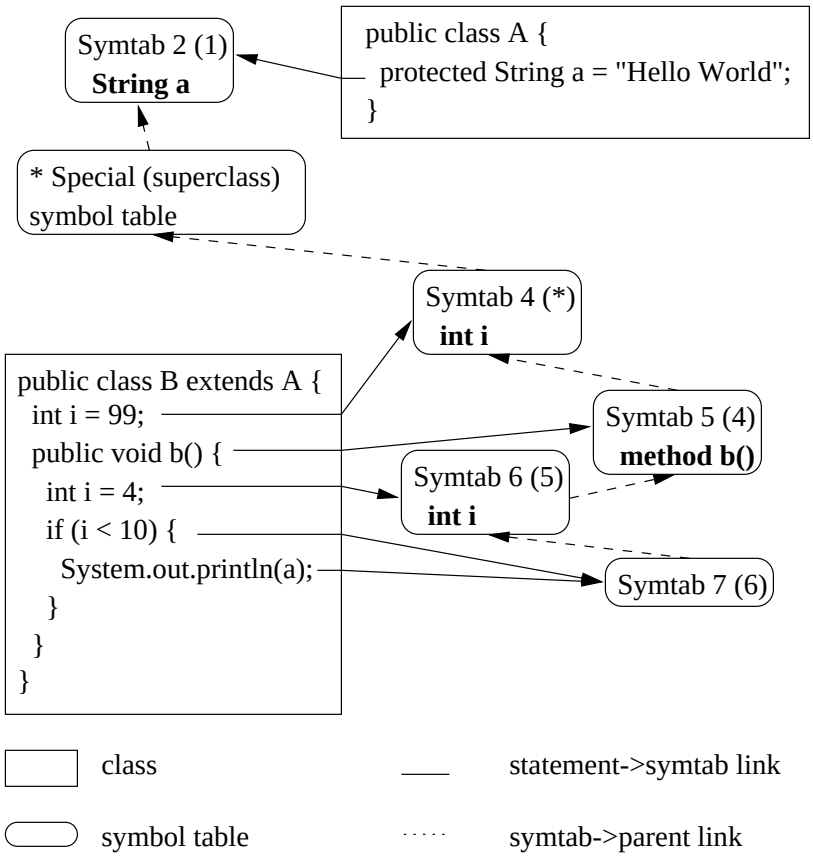


Figure 3.17: Symbol table example, demonstrating the use of **SuperSymbolTable** to represent inheritance.

3.10 Types

The type of a symbol or expression is represented by the **javaType** object. Types can be any of the primitive Java types, a reference type, `void`, or the `null` type. The `void` type only has meaning as a method return type, and the `null` type only has meaning as the type of an expression. The **javaType** object provides basic functionality for setting and examining the type represented, including a flag to indicate if the type is an array. A method is provided to obtain the *signature* of a particular type, as defined in the Java Virtual Machine Specification[7]. The various **javaType** types their Java equivalents, and their type signatures, are given in Table 3.8.

Static methods are provided to determine the result of applying binary or unary numeric promotion [4](5.6) to specified numeric types. When a the type is

javaType type	**Java type**	Type signature
t_byte	`byte`	`B`
t_short	`short`	`S`
t_char	`char`	`C`
t_int	`int`	`I`
t_long	`long`	`J`
t_float	`float`	`F`
t_double	`double`	`D`
t_boolean	`boolean`	`Z`
t_reference	`reference to Type`	`L`<class name>
t_void	`void`	`V`
t_null	`null`	<none>
t_invalid	<`none`>	<none>

Table 3.8: Type representation in HLIR

`t_reference`, the **javaType** object also includes the `String` name of the class referenced. This represents a reference to an object of the named Type. When an array type is represented, the number of array dimensions (but not their lengths) is included. The type `t_invalid` is the default value given to any symbol or expression, before the type evaluation pass executes. This type is also used for expressions which have no type, such as **naryExpression** and **emptyExpression**.

Chapter 4

Support Classes

In this chapter, we detail several classes which are essential to zJava and HLIR, but do not represent actual Java language elements.

4.1 The zJava Frontend

The zJava frontend uses the Java Tree Builder (JTB) [44] from Purdue University to automatically generate abstract syntax tree definitions based on the input grammar. The Java Compiler-Compiler (JavaCC) [45], from Metamata, is used to generate a parser which builds the AST defined by JTB. JTB also generates a Visitor design pattern, which provides the framework for the pass which traverses the AST and generates an HLIR representation.

The frontend lacks semantic checking and comprehensive error checking. Compilation stops when the first parse error is detected. In practice we use the standard Sun `javac` Java compiler to pre-process input programs and perform semantic checking. Only error-free programs are passed into the zJava frontend, where the intermediate representations are populated. It is a future goal of the project to implement semantic checking and error processing in the front end, and to implement byte code generation from HLIR.

4.2 zJavaUser

The **zJavaUser** class provides a well-defined place for users to place code for their compiler passes. A skeleton method is provided, which is invoked by the compiler after HLIR has been constructed.

The **zJavaUser** class provides two additional methods, one of which is invoked just before statements are added to a **stmtList**, and the other just after statements are removed. These methods provide a place for users to update the structures associated with their analyses, keeping them consistent under any transformations which modify the statement list.

4.3 zObject

At the root of the HLIR class hierarchy is the class **zObject**. Any object of type **zObject** can be cloned, converted to source code, and flagged as either owned or unowned. All HLIR class constructors check the ownership flags of **zObjects** passed into them. If the constructor expects to have ownership of a particular object, but finds that it is already owned, it will throw **zObjectOwnedError**.

Often an HLIR class only desires to have a "reference" to some object (that is, it expects some other entity in the program to own the object). An example of this is class **ifStatement**, which contains a reference to the **thenStatement** of its *then-part*. The **thenStatement** is owned by the statement list, rather than the **ifStatement**. In this example, the **thenStatement** must already be owned by some other object (that is, it must have already been added to the statement list) before it is passed into the **ifStatement**.

When a constructor expects an object to already be owned, but finds it to be unowned, it will throw **zObjectUnownedError**. In addition to performing these checks in the constructors, methods which modify the internal members of an HLIR object also perform ownership checks.

4.3.1 zAttribute

The **zObject** class also includes a list of **zAttribute** objects, which are used to store any attribute a user may wish to attach to an object in the representation. For example, compiler directives from the source code are encapsulated in attributes and attached to the appropriate objects. User-defined attributes can be created by deriving a new class from **zAttribute**.

A flag in the **zAttribute** class tells HLIR whether or not to convert each attribute into a compiler directive during source code generation, and another flag indicates which attributes should be included in during bytecode generation.

4.3.2 Source-to-Source Transformation

Every class derived from **zObject** must implement the method `toSource`. This method converts the object (and its members, recursively) into the equivalent Java source code, and outputs to a stream.

Generating source code from HLIR requires dealing with several special cases; as a result, source code is intended to be generated on a per-compilation-unit basis. That is, the `compilationUnit.toSource` method drives the process, and the appropriate method is recursively invoked on each object in the representation of the compilation unit. Trying to regenerate a method by iterating over its statements and invoking `toSource` in each one will generally lead to an incorrect representation in source code. This is clearly demonstrated by the fact that a `for` loop is flattened, with the *for-init* placed before the loop, and the *for-update* placed at the end of the loop body. Further, the special methods `<object_init>()` and `<clinit>()`, which are implicitly added and thus not included in source code generation, may contain initializer blocks which will need to be properly formatted and output (see §2.2.6 for details).

4.4 zLinkedList and zRefList

Thus far, only the ownership of individual objects by other HLIR objects has been discussed. However, much of the representation involves lists of objects, and this pro-

vides another mechanism for ownership enforcement. The classes **zLinkedList** and **zRefList** are derived from **java.util.LinkedList** for this purpose. These classes are based on the Collections framework used in the Polaris Internal Representation [5][6]. A **zLinkedList** may only contain objects of type **zObject**, and any **zObject** may be contained in at most one **zLinkedList**. The **zLinkedList** insertion methods first check that the object is an unowned **zObject**, then accept it into the list and mark it as owned. The removal methods perform the removal, then mark the object as unowned. The methods which provide access to elements of the list (such as those included in the `java.util.ListIterator` interface) verify ownership before returning the object.

The **zRefList** class exists for cases where objects in the list are expected to be owned by some other list. For example, a **switchStatement** contains a list of **caseStatement** objects, each of which is in fact owned by the enclosing statement list. Thus, **switchStatement** utilizes a **zRefList** to store its `case` statements.

Any object inserted into a **zRefList** must already be owned. Methods in **zRefList** which provide access to its elements first check that the element is still owned by another list. If an element in a **zRefList** is removed from the **zLinkedList** that owns it, any subsequent attempts to access that element through the **zRefList** will cause a **zObjectUnownedError**.

It is important to note that, in the absence of templates in Java, objects in a list are not required to be of any particular type, and elements of a list are not required to be of the same type. The **zLinkedList** and **zRefList** classes impose the constraint that all of their elements must be non-null and of type **zObject**.

4.5 zHashTable

Throughout the zJava HLIR, symbol tables are stored in the form of hash tables. Since every new scope has its own symbol table, there is no need to include declaration statements in the representation. When converting to source, the declaration statements can be built using the information in the various symbol tables. However,

since a sequence such as the following is legal:

```
int a = 6, b = a;
```

it is vital that the order of declarations be preserved. A standard hash table has no concept of insertion order. Thus, the **zHashTable** class was created, which gives each element a unique (and monotonically increasing) tag as it is inserted. This tag can be used to ensure that symbols are generated in the output source code in the order in which they were originally declared.

4.6 zClassLoader

It is anticipated that users will frequently want perform analysis on Java classes for which only the `.class` file is available, and not the source code. For example, an interprocedural analysis of a program which calls any of the standard Java class libraries is unlikely to have access to the source code for these libraries. This makes is desirable to have a way to integrate the information available in `.class` files into HLIR. The zJava HLIR is able to do this up to, but not including, the actual instruction opcodes inside methods. That is, there is no attempt to decompile bytecode into a high-level form. However, all other information in the `.class` file is represented in HLIR. The **zClassLoader** class, derived from **java.lang.ClassLoader** is used to load from `.class` files. The HLIR classes **ClassShell**, **MethodShell**, **zModifier**, and **javaType** include class constructors allowing them to be built from **java.lang.Class** and the classes in `java.lang.reflect`.

These classes also include constructors allowing them to be built from the zJava ByteCode Intermediate Representation (BCIR) [8]. BCIR includes support to store and read arbitrary attributes in the `.class` file (which would be ignored by a standard classloader).

4.7 zJavaException

The zJava HLIR includes a base exception class, **zJavaException**. This exception is raised in most cases where recovery from a problem within HLIR is expected to be possible. A base class for runtime errors, **zJavaError**, is also provided. This is raised in situations where recovery is not expected to be possible. The two classes **zObjectOwnedError** and **zObjectUnownedError**, are derived from this class to deal with errors specific to object ownership. These classes can be modified, or subclassed, if the exception model is extended in the future.

4.8 zDirective

The **zDirective** class is used to represent compiler directives, which provide a mechanism for passing information between the source code and HLIR. The general format of a zJava compiler directive is:

```
//zj <directive_type> <directive_body> ;
```

The valid directive types, and the syntax of the directive bodies, are specified in the modified Java grammar included with the zJava compiler distribution. In general, a directive is associated with the object it precedes. A directive placed after package and import declarations, and followed by an extra ";", is associated with the compilation unit. Multiple sequential directives are attached (in textual order) to the same object. Figure 4.1 shows examples of these and other directives. When HLIR is constructed, **zDirective** objects are encapsulated in **DirectiveAttribute** objects (a subclass of **zAttribute**), and inserted in the appropriate object's list of attributes.

Currently, zJava includes three types of directives. **StringDirective** allows provides a way to attach a simple text message to an HLIR object. **ExprDirective** is an example which allows any expression to be placed in a directive. The expression will be parsed by the frontend, and converted into the appropriate **zExpression** object, before being included in the directive. **DSVDirective** is used by the Data Structure Visualizer (DSV) example (see §5.3) to pass information to the DSV compiler pass.

```
package cow;
import spoo.fleem;

//zj Z_STRING "attached to compilationUnit";;

//zj Z_STRING "attached to classObject";
public class s {

  //zj Z_STRING "attached to varSymbol";
  double d;

  //  //zj Z_STRING "syntax error"
  {
    d = 12;
  }

  int Z_STRING;  // syntax error; directive types are
                 // keywords

  //zj Z_STRING "attached to methodShell";
  public void me() {

    //zj Z_STRING "attached to zStatement";
    d = 6.0;

    //zj Z_STRING "attached to varSymbol";
    int foo = 99;
  }
}
```

Figure 4.1: Examples of compiler directives.

In order to add a new directive, a user is required to:

- Define the syntax a new directive,
- Extend the grammar to recognize this syntax,
- Provide a class which the frontend will populate with the new directive, and
- Include in that class the functionality to convert the directive into source code.

The zJava compiler will then automatically populate the new directive classes during parsing, convert them into attributes and attach them to objects as HLIR is built,

and convert them back into source annotations during source code generation.

Chapter 5

Examples

This chapter details several examples of HLIR in use. We begin with a simple example showing the use of HLIR to insert a `for` loop into a method body. We then describe the zJava high-level control flow graph representation, implemented as a part of this thesis.

The final two examples are projects implemented by undergraduate students, and were invaluable sources of input towards the development of the zJava compiler infrastructure. In addition to verifying the functionality of the compiler and the completeness of the documentation, they demonstrated that the goal of rapid prototyping has been achieved. These examples led to the detection and correction of several bugs, and feedback from the users led to API improvements.

The first of these projects demonstrates the functionality of HLIR as a method of analysing and augmenting Java source code, adding code to user programs such that they interface with a data structure visualization library. The second demonstrates the use of HLIR to extract detailed data from Java source code, for use in software architecture visualization.

5.1 Statement Insertion

This simple example shows the insertion of a `for` loop into an existing method body. The source code equivalent of the loop is given below.

```
for (int i = 5; i < 10; ++i) {
  System.out.println("Hello World!");
}
```

The Java code given in Figure 5.1 demonstrates the HLIR calls necessary to construct and insert this loop. It is assumed that the variable `method_body` references the list of statements currently in the method, and that `idx` is the index into that list at which the new loop is to be inserted. The declaration of `i` will be translated into a symbol table entry, and this will be automatically merged with the existing symbol table information for the method body. Each use of `i` will be resolved to point to this symbol table entry, and the type of each expression will also be resolved.

5.2 Control Flow Graph

The package `hlir.zControlFlow` contains the high-level control flow graph (CFG) implementation of the zJava compiler. This implementation is built on top of HLIR, rather than being an intrinsic part of the core IR.

A control flow graph is composed of a set of *basic blocks*, linked by control flow edges [10]. A basic block is defined as a sequence of statements with only one entry and one exit point. An edge between two basic blocks represents a possible path of control flow between the two blocks in execution. Each basic block begins with a *leader* statement. Hence, identification of basic blocks is accomplished through the identification of leader statements in the program.

Control flow can be viewed as the combination of *lexical* and *non-lexical* flow. Lexical flow occurs when control flows from one statement to the statement lexically following it in the source code. When flow potentially jumps to a different statement, it is said to be non-lexical.

The overall structure of a control flow graph is contained in class **ControlFlowGraph**. This class represents the CFG for a single Java method. It includes a list containing all basic blocks, and provides the methods for creating the control flow graph of a method from HLIR. Each CFG includes two implicit nodes: `Entry`, which represents the entry point into the method (and has no predecessors), and `Exit`,

```
// Loop initializer:
javaType int_t = new javaType(javaType.t_int);
varSymbol loopVar = new varSymbol("i", int_t, new fieldModifier());
loopVar.setInitializer(new varExpression("5"));
LinkedList declsList = new LinkedList(); declsList.addLast(loopVar);
stmtList initList = new stmtList(new declStatement(declsList));

// Loop update
preUnaryExpression incrExpr =
  new preUnaryExpression(unaryExpression.op_plusplus,
                         new varExpression("i"));
stmtList updateList = new stmtList(new exprStatement(incrExpr));

// Loop conditional: i < 10
varExpression leftExpr = new varExpression(new String("i"));
varExpression rightExpr = new varExpression(new String("10"));
binaryExpression conditionalExpr =
  new binaryExpression(leftExpr, binaryExpression.op_lessthan, rightExpr);

// Loop body:
LinkedList argsList = new LinkedList();
argsList.addLast(new StringLiteralExpression("Hello World!"));
naryExpression arguments = new naryExpression(argsList);
varExpression methExpr = new varExpression("System.out.println");
methodInvocation invokeExpr = new methodInvocation(methExpr, arguments);
stmtList bodyList = new stmtList(new exprStatement(invokeExpr));

// Loop insertion:
forStatement forStmt = new forStatement(conditionalExpr);
stmtList newList = new stmtList(forStmt, initList, bodyList, updateList);
method_body.add(idx, newList);
```

Figure 5.1: Using HLIR to construct a `for` loop.

which represents control flow out of the method (and has no successors).

Each basic block is represented by a single **CFGNode** object. The **CFGNode** class contains a unique tag, references to the first and last statement in the basic block it represents, and lists containing references to all predecessor and successor blocks. Methods are provided to access and alter this information, including adding and removing forward and reverse control flow edges. Table C.7 of Appendix C (page 120) summarizes several interesting methods of class **CFGNode**.

A source-level representation does not contain explicit branch statements. The

generation of a high-level control flow graph involves the identification of leader statements, and the determination of the targets of non-lexical control flow edges. The following sections detail this process. The discussion of control flow due to exceptions, and the `try-catch-finally construct`, is left for §5.2.7. The figures presented in the following sections use solid edges to represent lexical control flow, and dashed edges to represent non-lexical control flow.

5.2.1 break/continue

The statement following a **breakStatement** or a **continueStatement** is always considered to be a leader statement. An unlabelled `break` causes the enclosing iteration statement or `switch` to terminate, and control flows to the statement after the **endStatement** of the terminated statement[1]. An unlabelled `continue` causes the enclosing iteration statement to terminate its current iteration and proceed to the next, and control flows to the **endStatement** of the loop (which represents evaluation of the conditional and the branch decision)[2].

Statements which have the potential to be the *target* of a `break` or a `continue` are not considered to be leader statements by default. There are too many statements which match this criteria and treating them all as leaders would unnecessarily fragment the CFG. They only need to be leaders if there is actually a `break` or `continue` statement which targets them. When a **breakStatement** or **continueStatement** is encountered while building the control flow graph, the target statement is calculated. If the target is not already a leader statement, the basic block containing the it is split.

Note that the control flow edges to the target may need to be *detoured* through one or more *finally clauses* (see §5.2.7.3).

[1] A labelled `break` statement causes the termination of the enclosing labelled statement whose label a matches that of the `break`. This statement need not be an iteration statement.

[2] A labelled `continue` statement causes the enclosing iteration statement, with a matching label, to terminate its current iteration and proceed to the next.

5.2.2 return

A **returnStatement** generally causes control to flow to the *Exit* node of the current control flow graph, representing termination of the method. The statement immediately following a **returnStatement** is always a leader statement.

If the `return` is enclosed by a *try block* or a *catch clause*, the control flow edges to `Exit` may need to be *detoured* through one or more *finally clauses* (see §5.2.7.3).

5.2.3 if-then-else

The statement lexically following an **ifStatement** (that is, the first statement of the *then-part*) is always considered to be a leader statement. As well, the *if-end* is a leader statement. Control flows from the *if-header* to the first statement of the *then-part*, and to the *if-end*. Control flows from the end of the *then-part* to the *if-end*.

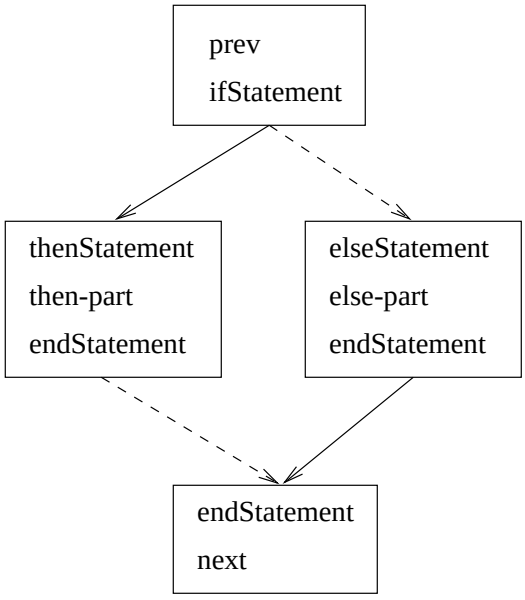


Figure 5.2: Control flow representation of an `if-then-else` construct.

When there is an *else-part* present, control flows from the *if-header* to both the beginning of the *then-part* and to the *else-header*. Similarly, control flows from the end of each of these parts to the *if-end*. This results in the control flow graph given in Figure 5.2.

5.2.4 switch

The *case-end*, and each *case-label*, are leader statements. The **blockStatement** (and corresponding **endStatement**), which contain the body of the `switch`, are orphaned; that is, they are in nodes which are not connected to any other node in the CFG. Their inclusion would merely complicate the construction and use of the CFG, and eliminating them does not discard any information.

The *switch-header* is linked to each *case-label*, and to the *switch-end*. The *switch-header* represents the comparison of the *switch-expression* to each *case-expression*. The *case-labels* themselves have no meaning in execution context. Each *case-end* is linked to the lexically following *case-label*, to represent case-case fallthrough. The last *case-end* is linked to the *switch-end*. These links are all formed while processing the *switch-header*. If any of the *case clause* contains a `break` or `continue`, the control flow will be updated when the `break/continue` is later processed.

The control flow model of a `switch` construct is given in Figure 5.3.

5.2.5 label

In a *labelled* construct, control simply flows lexically to the first statement of the labelled construct. Non-lexical control flow caused by `break` or `continue` statements targeting this label will be linked when such statements are processed.

5.2.6 Iteration Constructs

The first statement of a *loop-body* (that is, the statement lexically following a *loop-header* statement) is considered to be a leader statement. Similarly, the statement lexically following a *loop-end* is considered to be a leader statement. This leads to the control flow representation given in Figure 5.4. Note that the edge from the *loop-header* to `next` is not present for a `do` loop.

The *loop-end* statement represents the evaluation of the loop conditional expression, and deciding whether to branch back to the beginning of the *loop-body* or to the statement following the loop. In addition, for `for` and `while` loops, the *loop-*

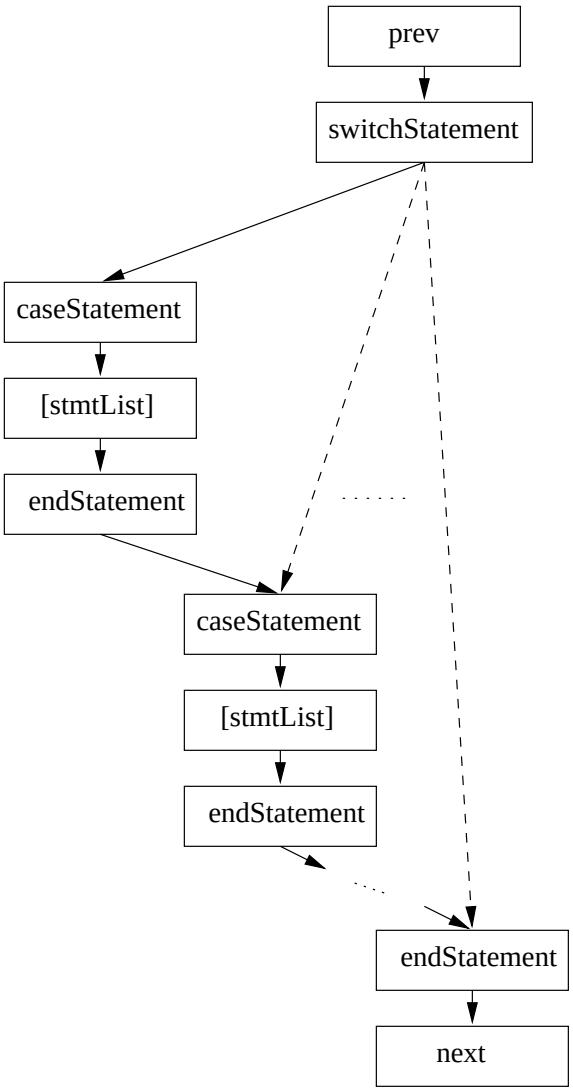


Figure 5.3: Control flow in a *switch* construct.

header statement represents the initial evaluation of the loop condition (since these two types of loops might never execute the body). Since the body of a `for` loop is flattened (see §3.5.3), there is no need for special control flow to represent the evaluation of initializer or update. The *for-init* code comes before the *loop-header*, and the *for-update* statements are part of the *loop-body*.

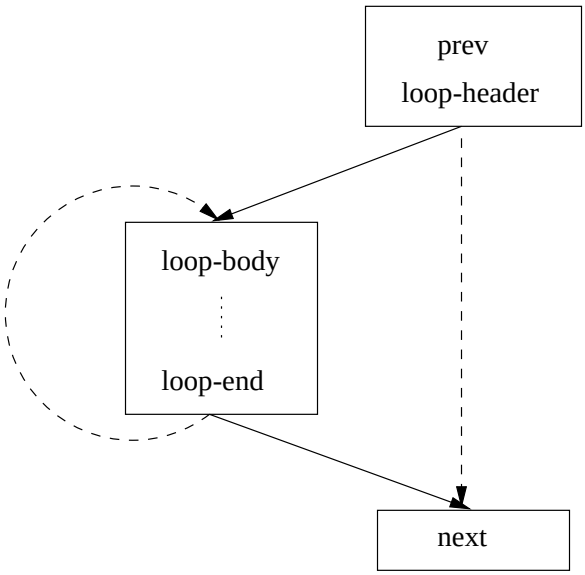


Figure 5.4: Control flow in an *iteration* construct.

5.2.7 Exceptions

The presence of exceptions, and exception-handling constructs, significantly complicates the control flow of a program [11]. We begin by describing the Java exception model, and the general exception flow of a `try-catch-finally` construct. We then describe the complications introduced by expressions which may raise exceptions, and detail how the HLIR control flow representation models exception flow.

5.2.7.1 Java Exception Model

Java exceptions are *precise*; thus, when an exception is thrown at a program point, the following must hold true:

- All effects of statements and expressions from before the exception must appear to have occurred.
- Any effects resulting from the speculative execution of statements and expressions following the exception must not be visible.

When an exception is raised, it causes a transfer of control to either a matching handler block of the method, or to the exit block of the method. The user-visible

state after an exception is thrown depends on whether or not the exception *escapes* (is not caught by any handler blocks) the method. If an exception is caught by a handler block in the method, local variables of the method are visible to the handler. If the exception escapes, the handler which eventually catches it does not have access to this information.

Java exceptions fall into four categories:

- *Checked exceptions*. These are always explicitly raised by a `throw` statement, and declared in the `throws` clause of any method from which they may potentially escape.
- *Runtime exceptions*. These are unchecked exceptions which ordinary programs may wish to catch. These may be raised by any of several instructions, and are not necessarily declared in `throws` clauses of methods.
- *Errors*. These are exceptions from which ordinary programs are not expected to recover.
- *Asynchronous exceptions*. These are exceptions caused by an error in the Virtual Machine.

Any precise control flow model must ensure that the first two categories are correctly represented. Control flow due to errors and asynchronous exceptions is not normally considered.

5.2.7.2 Try-Catch-Finally

The **catchStatement** of each *catch clause* is considered to be a leader statement, as is the **finallyStatement** heading the *finally clause*. Additionally, the **endStatement** terminating the *try* construct (the *try-end*) is considered to be a leader statement. Control flows from the end of the *try block* to each *catch clause*, to the *finally clause*. In the absence of a *finally clause*, control also flows to the *try-end* (to represent normal completion of the *try block*). Additionally, a forward edge is added from the end of each *catch clause* to the beginning of the *finally clause* if present, otherwise to the

try-end. Finally, control flows from the end of the *finally clause* to both the *try-end* and `Exit` (the latter is required to represent abnormal completion of the *finally clause* itself). The control flow involved in a `try` construct is shown in Figure 5.5.

Note that the control flow edges to `Exit` mentioned in the above may be detoured if the `try` construct is nested within another *try-block* or *catch clause* (see §5.2.7.3).

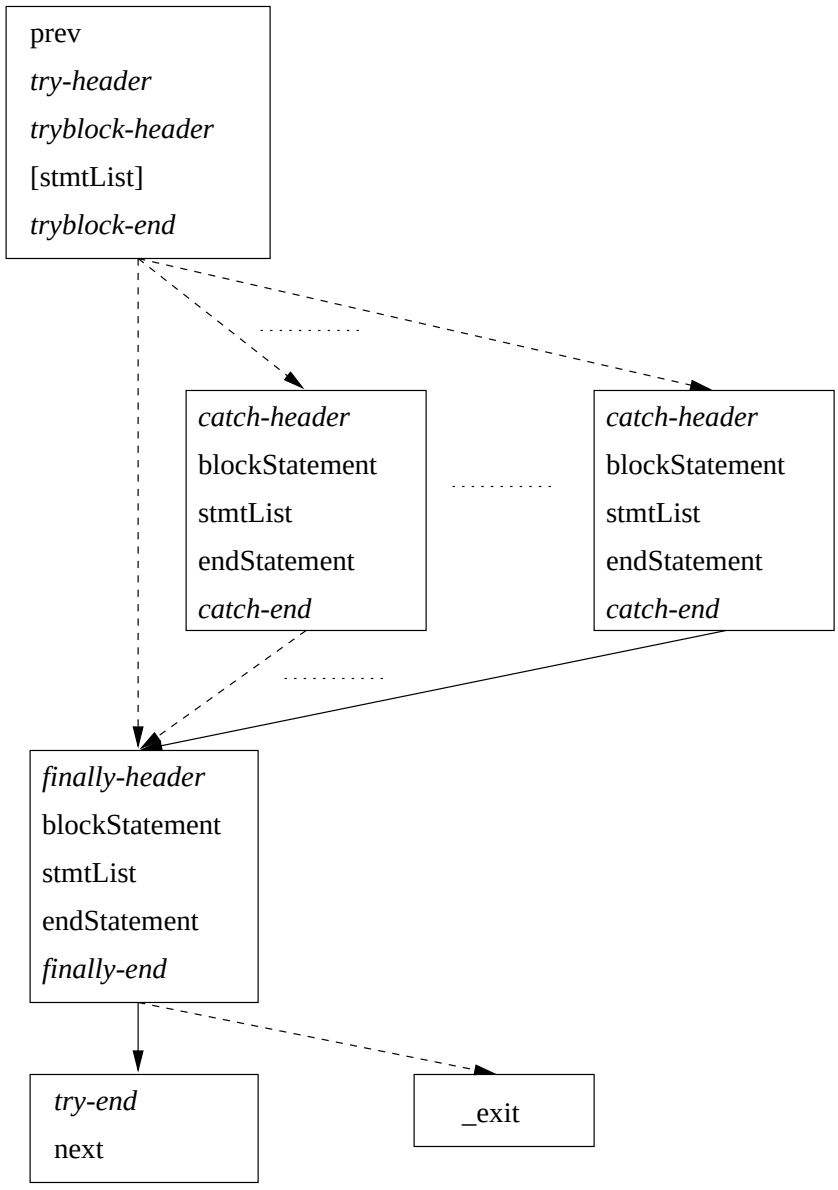


Figure 5.5: Control flow in a *try* construct.

5.2.7.3 Propagating Jumps

There are four types of *jumps* in Java; the `break`, `continue`, `return`, and `throw` statements. Unlike a *branch* statement, a jump may cause control to transfer to a statement outside the current construct. When control jumps out of a *try block* or a *catch clause*, it is necessary to detour control through the associated *finally clause*, if present, before continuing to the jump target [4, §14.19.2][3].

The algorithm for propagating jump statements proceeds as follows:

- Initialize `source` to reference the jump statement.
- Initialize `encloser` to be the construct directly enclosing the jump statement.
- Repeat, until the construct that is the target of the jump is encountered:
  - Examine `encloser`. If it is a *try block* or a *catch clause*, and there is an associated *finally clause*:
    - Add a control flow edge from `source` to the beginning of the *finally clause*.
    - Set `source` to reference the end of the *finally clause*.
  - Set `encloser` to be the construct directly containing the current `encloser`.

In this way, a jump from within a nest of `try-catch-finally` constructs (i.e. a `try-catch-finally` which is within a *catch clause* of another `try-catch-finally`) is correctly propagated.

The **ControlFlowGraph** class maintains two mappings to facilitate this process—one from each statement to the enclosing statement, and one from each statement to the basic block containing it.

[3]While it is clear that the *finally clause* executes regardless of how the *try block* completes, the Java Language Specification [4] is ambiguous on the impact of abrupt completion of a *catch clause*. §14.19.2 states that when a *catch clause* completes abruptly, the *finally clause* is executed. However, the sections detailing each type of jump statement (§14.14-14.17) specifically mention their behaviour within a *try block*, but fail to mention *catch clauses*.

5.2.7.4 Potential Exception-Causing Instructions

Many statements in Java have the potential to cause exceptions. The most obvious example is the `throw` statement, which explicitly raises an exception. The semantics of `throw` statements, and the the `try-catch-finally` construct, are complex in themselves [11]. Further, several other common statements have the potential to raise an exception, such as array access, field access, method invocation, variable read or write, and object creation. The term *potential exception-causing instruction* (PEI), as introduced by the Jalapeno project [12][46], can be used to describe such statements.

The frequency of PEIs can degrade the usefulness of a traditional CFG. A PEI can cause the flow of control to change; thus it is considered to terminate a basic block. This kind of model would result in very many, very small basic blocks, and a graph with many edges. The effectiveness of local analyses is greatly reduced when basic blocks are small; and the time complexity of any analysis increases as the number of nodes and edges increases. Control flow analysis is further complicated by the need to perform type-matching to determine which handler blocks can potentially be targets of an exception. In the case of a `throw` statement, or a method invocation, the run-time type of the raised exception may be a subclass of the declared type, necessitating flow-sensitive type propagation to determine the matching handlers.

We base out modelling of exception on the *Factored Control Flow Graph* from the IBM Jalapeno project [12]. We explicitly model exception flow only in the case of a `throw` statement, grouping all PEI edges into a single *factored edge* at the end of an *extended basic block* [10]. An extended basic block is defined as a basic block with a single entry point and multiple exit points. This allows the granularity of basic blocks to remain at a useful level, and keeps the complexity of the graph under control. Explicit exception flow edges are added for all `throw` statements. If the **throwStatement** is not enclosed by a `try` construct, an edge is simply added to `Exit`[1]. If the **throwStatement** is within a *try block*, we follow the most conservative approach and include an edge to the **endStatement** of the *try block*, which is then linked to every *catch clause*, and either the *finally block* if present, or `Exit`[1]. If a

throwStatement is inside a *catch clause*, all that is required is to add an edge to either the *finally clause* if present, else to `Exit`[1].

5.2.7.5 Precise Modelling of Exceptions

Precisely modelling exception flow in Java is a complicated task. A precise model would involve determining which, if any, *catch exception variables* match the type of each exception thrown, and adding edges only to the applicable *catch clauses*. Thus, the possible exceptions raised by each PEI must be known, and their assignability to each potential handler's *catch exception variable* must be determined. Since conversions and promotions of reference types are not implemented in HLIR, this determination cannot be made. In the case of method invocations, determining possible exceptions becomes an interprocedural analysis. While any checked exceptions thrown must be declared in the method declarator, runtime exceptions need not be specified. Any method containing PEIs (effectively, every method) has the potential to raise a runtime exception and pass it back to the caller. The analysis is further complicated by the fact that exceptions passed out of a *finally clause* may have originated in the *try block*, any of the *catch clauses*, or the *finally clause* itself.

The CFG representation in HLIR is conservative—that is, the lack of precision results in extra edges, rather than the omission of any control flow edges. Our representation can be readily extended to a precise model, similar to the FCFG, if precise modelling of PEI exception semantics is required in the future.

5.3 Data Structure Visualizer

The Data Structure Visualizer (DSV) [47] was a summer project in which one goal was to demonstrate the functionality of the zJava compiler infrastructure. DSV provides a means of graphically representing the objects and pointers in a program. It is is intended for use as a teaching and debugging tool, dynamically displaying common

[1] Again, if the statement is nested within another *try block* or *catch clause*, control may be detoured (see §5.2.7.3).

data structures (linked lists, binary trees, etc) as a program executes.

This project involved creating an HLIR pass to automatically augment Java programs to interface with an existing implementation of DSV, which combined the TCL Nodes Graphics Library (NGL) and the C++ ScreenManager utility [48]. Interfacing with the existing DSV implementation meant encapsulating its classes in wrappers which could be invoked through the Java Native Interface[4] (JNI), and then writing a zJava compiler pass to add calls to these classes where appropriate.

The zJava compiler pass which augments the user program, called the DSV Setup Tool [47], makes use of many of the functions provided by HLIR:

- Compiler directives are placed in the user source, to inform the compiler which classes represent a "Node" (of a linked list, doubly linked list, or binary tree), which fields in the Nodes represent pointers, and which classes contain data structures making use of the Node class. zJava was extended to recognize these new compiler directives as they became needed.

- Static initializer blocks are added to all classes which must be drawn on the screen (to register them with the ScreenManager), and additional statements are added to these blocks at a later stage.

- The main() method is identified and augmented to initialize the ScreenManager.

- User classes are augmented to call DSV when Nodes or references to Nodes are created or modified, to change what is on the screen. This requires examining each statement that might represent object allocation or an assignment, and constructing a DSV call based on the gathered information.

- A method is added to each Node class, to help handle garbage collection within DSV. This makes use of the symbol table methods to identify when Node references go in and out of scope.

- The augmented program is converted into Java source, which is then compilable by a standard Java compiler.

[4] http://java.sun.com/j2se/1.3/docs/guide/jni/

The code in Figure 5.6 illustrates the use of HLIR to identify statements in the user code which create new node objects, and to insert a DSV library call to register the new object and draw it on the screen.

```
// Assuming "stmts" is an iterator over the method body statements.
symboltable cur_SymTab = ((zStatement)stmts.next()).symboltable();

// Get a list of symbols which came into scope with this statement
ListIterator genIter = cur_SymTab.genSet(meth_sym_tab).listIterator();
while(genIter.hasNext()) {
  varSymbol newSym = (varSymbol) genIter.next();
  if((newSym instanceof varSymbol) && (newSym.type().isType_reference())
    && (Directive_Table.containsKey(newSym.type().name()))
    && (Directive_Table.get(newSym.type().name())
        instanceof classObject))
  {
    // Add a statement to create a variable to store the symbol's
    // DSV address.
    declStatement addrDecl =
      createAddrDecl(newSym.name() + "_addr", mainClass, this_class);
    stmts.add(new stmtList(addrDecl));

    // Add a statement to call DSV and draw the symbol on the screen.
    methodInvocation newNodeExpr = createNewNodeExpr("createNewNode",
      newSym.name() + "_addr", "DT_POINTER_VARIABLE",
      "\"" + new_Var.name() + "\"", newSym.type(),
      mainClass, this_class);
    stmts.add(new stmtList(new exprStatement(newNodeExpr)));
  }
}
```

Figure 5.6: Using HLIR to add DSV library calls each time a node is allocated.

5.4 Software Architecture Visualization

The Portable Bookshelf (PBS) [49], developed at the University of Toronto, is an implementation of the Software Bookshelf [50], a web-based framework for the presentation and navigation of information representing large software systems. The PBS system is used to generate landscapes of each subsystem, displaying objects within the subsystem and the relationships between them. The decomposition of

the system into subsystems, the granularity of the objects analysed, and the set of possible relationships, can all be controlled.

The first stage in the creation of a software landscape is the design of a language model, or high-level schema. This consists of the set of entities and relations to be represented in the landscape. Next, a source-level schema must be designed, to determine the set of facts which must be extracted from the source code. PBS makes use of the tuple-attribute form (TA) [51] as the syntactic representation for source models. TA is text-based format which contains both schemas and extracted facts. Examples of the kind of low-level facts (specific to Java) which must be extracted from the source code are given in Table 5.1.

Relation	**From (A)**	**To (B)**	**Description**
typeDefBy	Type	Sourcefile	Type **A** is defined by source file **B**.
methodDefBy	Method	Type	Method **A** is defined by **B**.
castsTo	Method	Type	Method **A** casts an expr to type **B**.
arrayType	Type	Type	Array **A** is an array of type **B**.
calls	Method	Method	Method **A** calls method **B**.

Table 5.1: A subset of the low-level relation types for Java [52].

Once a language model is defined, a landscape is generated as follows:

- Extract the source-level facts, into a TA factbase.
- Manipulate the factbase with `grok`[3], producing the high-level facts.
- Use `lslayout`[3] to generate landscapes from the high-level facts.
- Display the landscape with `lsedit`[3].

We base our landscape generator on Ivan Bowman's `bx` suite[5]. We replace the first step with a zJava compiler pass [53] which uses HLIR to extract source-level facts from the Java program to be modelled. The extraction pass requires detailed information at even the lowest level of the source code, including the type and scope of each symbol. The HLIR API provided convenient access to all of the required information.

[3]See http://swag.uwaterloo.ca/pbs/tools.html for a description of these tools.

[5]http://plg.uwaterloo.ca/~itbowman/pub/bx-1.0.zip

For example, the `methodDefBy` relation requires information about what methods are defined in a class, which is directly available from the symbol table maintained in the representation.

Chapter 6

Conclusions

The development of the zJava HLIR has shown that it is feasible to create a source-level representation for Java code, as a robust framework for the analysis and transformation of Java programs. We have included a precise representation of symbol scopes and the resolution of symbol and expression types. We incorporate information from classes not available in source form, through the seamless integration of `.class` files into the representation. We represent code which, according to the Java Language Specification [4], must implicitly be added by a compiler when not present in the source code. We also include a control flow representation, using a design which minimizes the effect of exceptions on the complexity of the graph.

Our design has placed an emphasis on ease-of-use and rapid development of new compiler passes. Through the use of robust error checking and compile-time error reporting, we have a system which enforces its internal consistency as well as the syntactic consistency of the programs it represents, and which detects incorrect uses of the IR as early as possible in the development process.

We have created a set of support classes to implement the concept of object ownership, and prevent the sharing of objects. Our system is designed to be extensible, to allow users to build on the functionality provided. Annotations can be used to pass information between source code, HLIR, and bytecode. Hooks are provided for users to insert code to incrementally update their own structures whenever a statement list is modified.

Within the context of the zJava compiler project, HLIR will serve as the basis for analyses such as call graph and data dependence graph construction, leading towards the ultimate goal of automatic parallelization.

We have already used HLIR as a program analysis and augmentation tool in two example compiler passes, demonstrating that it is both functional and correct.

6.1 Future Work

6.1.1 HLIR-BCIR links

A zJava compiler pass to implement bytecode generation [9] is one obvious extension to our work. This pass would transform HLIR into BCIR, which can then be output as `.class` files. The interaction between HLIR and BCIR can include the ability to add user-defined attributes to the bytecode. These attributes can be generated by a zJava pass, or specified in the source code as compiler directives. A further future goal is the inclusion of additional links between the high- and low-level representations, potentially allowing them to remain consistent with one another under transformations, and allowing compiler analyses to affect both levels simultaneously.

6.1.2 Semantic Consistency

While semantic verification, as a single complete pass over the IR, could easily be implemented, we are unsure what impact automatic semantic consistency maintenance would have on the efficiency of HLIR. While speed is not a driving goal of the project, the overhead of incremental semantic consistency could be unacceptable.

6.1.3 Control Flow

Our control flow representation currently does not model exceptions other than those explicitly raised by a `throw` statement. A possible future direction would be the inclusion of special edges to represent flow from *potential exception-causing instructions* [12], such as method invocations, pointer dereferences, and array accesses. An-

other goal is to extend HLIR to automatically update control flow information when a statement list is modified.

6.1.4 Java1.1

We currently do not model several of the language elements introduced with Java 1.1 [13]. This includes nested class declarations, anonymous array and class declarations, *outer this* expressions, certain types of array initializers, some forms of the `Type.class` expression, and the impact of nested classes on scope information. It is a future goal of zJava to represent the extended syntax of Java1.1.

6.1.5 Builder Methods

One of HLIR's most useful features is the ability to create new Java statements and add them to user programs. Currently, constructing even a simple statement requires several steps. First, a user must obtain references to any symbols to be included, and use these to construct the expressions of the statement. The statement is then created, using these expressions. The statement is then placed in a new statement list, which is finally added to the method body.

Development time could be significantly reduced if these steps could be combined and simplified. A set of helpful methods, similar to the SUIF Builder Library [16], would be an invaluable addition to HLIR.

6.1.6 Support Classes

Our current implementation of **zLinkedList** (and **zRefList**) is an extension of `java.lang.util.LinkedList`. While elements of a Java list can be of any reference type, we impose the constraint that all elements must be assignable to type **zObject**. A side effect is that our implementation of lists is dependent on internal details of the implementation of `java.lang.util.LinkedList`, and HLIR may not function correctly with a different set of standard library classes. This problem stems from the fact that a fundamental property of a `LinkedList` is that is is unaware

of (and uninterested in) the type of each object it stores. The HLIR **zLinkedList** class, however, places restrictions on the type of objects it may contain. Many of the methods in **zLinkedList** which override `LinkedList` methods simply add some functionality and then invoke the superclass version of the method. In certain cases the superclass methods invoke other methods of the class, which will result in the invocation of another **zLinkedList** method. Depending on the implementation of `LinkedList`, these cross-invocations may perform a cast on the argument, causing the **zLinkedList** method to generate an error.

Part of the future work planned for zJava is the re-implementation, from scratch, of these list classes, and the inclusion of other ownership-aware collection classes.

6.1.7 Type Conversions

The zJava HLIR currently does not include support for type conversions and promotions [4, §5], with the exception of unary and binary numeric promotion. This means that we are unable to determine the assignability of one reference type in another. This makes it impossible to perform certain aspects of type evaluation (§3.6.2.14), as well as several aspects of semantic verification. In the future, we would like to extend HLIR to support all forms of type conversion and promotion.

Appendix A

Testing and Verification

In this appendix we describe several of the procedures used during the development of the zJava infrastructure to verify its functionality and correctness. Since zJava is intended as a research infrastructure, it can be difficult to measure its functionality. We have found that the best feedback comes from having others use zJava to develop compiler passes, as detailed in Chapter 5. Putting HLIR to use has led to the correction of many bugs, and enhancements to the API. The zJava compiler project is ongoing; further uses of HLIR as a research tool will inevitably lead to additional corrections and enhancements.

The processes described below focus on the correctness of our source-to-source transformations. These tests verify that HLIR can handle the input source (that is, parse the code into HLIR), and that the regenerated source code has the same semantics as the original.

A.1 Self-Compile Test

During the development of HLIR, a self-compile test was used as a baseline to verify the correctness of the system.

The self-compile process follows these steps:

1. Source-to-source transform all HLIR classes. This step verifies that HLIR can parse, represent, and regenerate a large, complex program which makes use of

most aspects of the Java language.

2. Compile resulting source files (with a standard Java compiler). This step proves that HLIR source code generation outputs compilable code.

3. Use the newly compiled version of HLIR to again source-to-source transform the original code.

4. `diff` the source code generated in step 1 with that generated in in step 3. This step proves that the source-to-source transformation does not alter the program.

It is important to point out that between the first and second steps, we patch a few files, to overcome the fact that HLIR ignores nested classes.

A.2 SPEC Benchmarks

We have also tested the correctness of our source-to-source transformation using the source code included with the SPECjvm98[1] benchmarks. We limited out test to the top level of the source tree for each benchmark. We use a process similar to the self-compile test:

1. Run the original benchmarks, to generate baseline output data.

2. Source-to-source transform the available `.java` files with `zJava`.

3. Compile the resulting source files with `javac`.

4. Run the newly compiled benchmarks.

5. Compare the output data from step 1 with that from step 4.

We have found that the output is identical for every benchmark, verifying that `zJava` does not modify the program semantics. Again, it is important to note that some of the benchmark source files do not source-to-source transform properly, due

[1] http://www.spec.org/osg/jvm98/

to the use of some Java1.1 language elements. We patch these files manually during testing.

A.3 Lines of Code

We have implemented a simple HLIR pass which counts the number of executable statements in a program. While this is not a robust test of HLIR's correctness, it was useful in estimating the problem size of our other tests. This pass counts every statement type except the following:

- **thenStatement**
- **elseStatement**
- **labelStatement**
- **blockStatement**
- **tryBlockStatement**
- **emptyStatement**
- **endStatement**
- **forInitStatement**
- **forUpdateStatement**

HLIR consists of 106 source files, comprising of approximately 27 000 lines of text. Using the pass described above, we find that HLIR consists of approximately 8 700 "lines of code". Note that much of the difference is due to the presence of `javadoc` comments in the source; for short methods, the comment block can be several times the length of the method itself! The tested SPECjvm98 files consisted of about 11 000 lines of text, containing approximately 4 800 "lines of code".

Appendix B

The zDebug Interface

A debugging interface has been designed to help users understand how the source code translates into HLIR, and to provide a mechanism for testing various functions of HLIR. The `zdb` interface is a very fast and easy way to examine the effect of a compiler pass on the representation. The `zdb` interface, implemented in the **zDebug** class, allows users to traverse a populated HLIR, stepping into objects and examining their contents. It allows access down to the symbol level, and includes support for resolving the type of an expression (as described in §3.6.2). The ability to interact with the representation in this manner was a key element in the development of HLIR itself.

Any HLIR class which is to be accessed by `zdb` implements the interface **zDebuggable**. This includes providing implementations of handler methods which are called by `zdb` when an object of that type is encountered. This design makes it very easy to add new functionality to the debugger.

There are five "levels" in `zdb`: *compilation unit*, *class*, *method*, *statement*, and *expression*. When `zdb` is invoked, traversal begins at the compilation unit level. After each command, a text representation of the HLIR object currently being examined is output, followed by a prompt. At each level various commands are available, as defined by the handler for that level. At all times, the following are possible:

- `prev` : step back to the previous object

- `next` : step forward to the next object (the same as just hitting `<enter>`)
- `up` : jump up to the level above
- `symbols` : display the current symbol table
- `all_symbols` : display the current branch of the tree of symbol tables, starting with the current table and traversing to the top-level (class) symbol table, and then display available superclass symbols.
- `help` : display available commands (as defined by a handler for the current level)

Additional commands available at each level are summarized in Tables B.1 - B.5.

Command	**Action**
`imports`	list the import declarations of this compilation unit
`classes`	step down into the *class* level

Table B.1: Additional commands available at the *compilation unit* level.

Command	**Action**
`fields`	list the class and instance variables of this class
`methods`	step down into the *method* level

Table B.2: Additional commands available at the *class* level.

Command	**Action**
`fargs`	list the formal arguments of this method
`symbol`	examine the **methodSymbol** for this method
`stmts`	step down into the *statement* level

Table B.3: Additional commands available at the *method* level.

The **zDebug** class also provides a special method `zPrint`, which is used by various HLIR methods to print debug information. Each class which implements **zDebuggable** contains a flag indicating whether or not `zPrint` should output the debug

Command	Action
`exprs`	step down into the *expression* level

Table B.4: Additional commands available at the *statement* level.

Command	Action
`type`	display the type of this expression
`resolve`	perform type-evaluation on this expression
`stmt`	display the statement to which this expression belongs

Table B.5: Additional commands available at the *expression* level.

messages specific to that class. It is a future goal of HLIR to provide more fine-grained control of debugging output.

Appendix C

zJava HLIR API

This appendix summarizes several of the interesting methods of various HLIR classes. Complete API documentation is included with the zJava compiler release.

Method name	**Summary**
`addClass(ClassShell)`	Add a class to this compilation unit.
`addImport(String)`	Add an import declaration to this compilation unit.
`program()`	Get a reference to the **Program** object containing this compilation unit.
`classes()`	Get the list of classes and/or interfaces declared in this compilation unit.
`imports()`	Get the list of imports declarations for this compilation unit.
`symtab()`	Get a reference to this compilation unit's symbol table, which contains an entry for each class or interface declared.
`findAccessibleClass(String)`	Search all accessible sources (including `.class` files) for a matching class.

Table C.1: Several methods of the **compilationUnit** class

Method name	Summary
`addClassInitializer(stmtList)`	Add a new static initializer block to this class.
`addInstanceInitializer(stmtList)`	Add a new instance initializer block to this class.
`addField(varSymbol)` `removeField(varSymbol)`	Add/remove a field in this class.
`addMethod(methodObject)` `removeMethod(methodObject)`	Add/remove a method in this class.
`methods()`	Get the list of methods declared in this class.
`getFields()`	Get the list of class/instance variables declared in this class.
`modifiers()`	Get the access modifiers applying to this class
`getSymbol()`	Get the **classSymbol** for this class.
`symtab()`	Get the symbol table for this class, containing symbols for each method and class/instance variable declared.

Table C.2: Several methods of the **classObject** class, including some inherited from **ClassShell**.

Method name	Summary
`stmts()`	Get the statement list containing the body statements of this method.
`symtab()`	Get the symbol table for the topmost scope of this method, containing symbols for this method's parameters.
`enclosingClass()`	Get a reference to the **classObject** containing this method.
`signature()`	Get the method signature which uniquely identifies this method within its declaring class.
`getSymbol()`	Get the **methodSymbol** for this method.
`modifiers()`	Get the access modifiers applying to this method.

Table C.3: Several methods of the **methodObject** class, including some inherited from **MethodShell**.

Method name	Summary
`expressions()`	Get a list of the expressions directly contained in this statement.
`allExpressions()`	Get a list of all expressions contained in this statement, including subexpressions.
`enclosingStatement()`	Get the header statement of the Java construct containing this
	statement.
`getList()`	Get a reference to the **stmtList** containing this statement.
`statements()`	Get a list of every statement referenced by this statement.
`next()`	Get the statement following this one in the statement list.
`prev()`	Get the statement preceding this one int the statement list.
`symtab()`	Get the symbol table for the scope containing this statement.

Table C.4: Several methods of the **zStatement** class

Method name	Summary
`addAll(int, stmtList)`	Inserts all of the elements of the specified list into this list, starting at the specified position, in order.
`getMethod()`	Get the method containing this list.
`listIterator(idx)`	Obtain a list iterator over the elements of this list, starting at the specified position.
`loopIterator(idx)`	Obtain a list iterator over all loop headers in this list, starting at the specified position.
`statementIterator(idx)`	Obtain a list iterator over all statement in this list, starting at the specified position, which match the specified mask.
`remove(int)`	Remove the statement or construct at the specified position, if permitted.
`remove(Object)`	Remove the specified statement or construct, if permitted.

Table C.5: Several methods of the **stmtList** class

Method name	Summary
`subexpressions()`	Get a list of any expressions contained in this expression.
`allSubExpressions()`	Get a list of all expressions contained in this expression, including subexpressions.
`allSymbols()`	Get a list of all symbols contained in this expression.
`evaluateType(javaType)`	Perform type-evaluation on this expression.
`getStmt()`	Get the statement containing. this expression.
`type()`	Get the type of this expression.

Table C.6: Several methods of the **zExpression** class

Method name	Summary
`setLast(zStatement)` `getLast()`	Set/get the first statement in this basic block.
`setFirst(zStatement()` `getFirst()`	Set/get the last statement in this basic block.
`addForwardFlow(CFGNode)` `removeForwardFlow(CFGNode)`	Add/remove a forwards control flow edge between this basic block and the one specified.
`addReverseFlow(CFGNode)` `removeReverseFlow(CFGNode)`	Add/remove a reverse control flow edge between this basic block and the one specified.
`clearForwardFlows()` `clearReverseFlows()`	Clear all forwards/reverse control flow edges from this basic block.
`splitNode(zStatement)`	Split this basic block, starting a new basic block with the specified statement.

Table C.7: Several methods of the **CFGNode** class

Bibliography

[1] D. Padua, R. Eigenmann, J. Hoeflinger, P. Peterson, P. Tu, S. Weatherford, and K. Faigin, "Polaris: A new-generation parallelizing compiler for MPPs," Tech. Rep. CSRD-1306, University of Illinois at Urbana-Champaign, June 1993.

[2] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis, "An overview of the SUIF2 compiler infrastructure."

[3] C. D. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten, "The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran," in *Languages and Compilers for Parallel Computing, 1989 Workshop*, (Urbana, Ill.), pp. 423–453, Cambridge, Mass.: MIT Press, 1989.

[4] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, 1996.

[5] K. A. Faigin, J. P. Hoeflinger, D. A. Padua, P. M. Petersen, and S. A. Weatherford, "The Polaris internal representation," Tech. Rep. CSRD-1317, University of Illinois at Urbana-Champaign, February 1994.

[6] K. A. Faigin, "The polaris internal representation," Master's thesis, University of Illinois at Urbana-Champaign, 1994.

[7] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[8] D. Lew, "BCIR: A framework for the representation and manipulation of Java bytecode," Master's thesis, University of Toronto, 2000.

[9] V. Quan, "Bytecode generation in the zjava compiler," 2000. Undergraduate thesis, University of Toronto.

[10] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[11] S. Sinha and M. J. Harrold, "Control-flow analysis of programs with exception-handling constructs," Tech. Rep. OSU-CISRC-7/98-TR25, The Ohio State University, July 1998.

[12] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for the analysis of Java programs," in *Proceedings of the ACM SIGPLAN–SIGSOFT workshop on Program analysis for software tools and engineering*, 1999.

[13] K. Arnold and J. Gosling, "Changes for Java 1.1," 1996. http://www.javasoft.com/docs/books/jls/html/1.1Update.html.

[14] "Inner classes specification," 1997. http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses.

[15] "Clarifications and amendments to the inner classes specification," 1998. http://www.javasoft.com/docs/books/jls/nested-class-clarify.html.

[16] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy, *An Overview of the SUIF Compiler System.* Computer Systems Lab, Stanford University.

[17] A. Duncan, B. Cocosel, C. Iancu, H. Kienle, R. Rugina, U. Hlzle, and M. Rinard, "OSUIF: SUIF 2.0 with objects," in *Proceedings of the Second SUIF Compiler Workshop*, 1997.

[18] U. o. C. S. B. OSUIF Group, Department of Computer Science, "The OSUIF library," 1999. http://www.cs.ucsb.edu/~osuif/.

[19] G. E. Weaver, K. S. McKinley, and C. C. Weems, "Score: A compiler representation for heterogeneous systems," in *Proceedings of Heterogeneous Computing Workshop*, 1996.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 39, July 1987.

[21] R. Cytron, J. Ferrante, B. K. Rosen, and M. N. Wegman, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, October 1991.

[22] F. Bodin, P. Beckman, D. Gannon, J. Botwals, S. Narayana, S. Srinivas, and B. Winnicka, "An object-oriented toolkit and class library for building Fortran and C++ restructuring tools," in *Object Oriented Numerics Conference (OONSKI)*, 1994.

[23] D. Gannon, "Sage++ users guide: A class library for building Fortran 90 and C++ restructuring tools," 1993. http://extreme.indiana.edu/sage/sagexx_ughtml/sagexx_ug_toc.html.

[24] A. Chien, J. Dolby, B. Ganguly, V. Karamecheti, and X. Zhang, "High level parallel programming: the illinois concert system." Submitted for publication, 1998.

[25] J. B. Plevyak, *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, August 1996.

[26] J. W. Davidson and C. W. Fraser, "Code selection through object code optimization," *ACM Transactions on Programming Languages and Systems*, vol. 6, pp. 505–526, October 1984.

[27] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers, "Vortex: An optimizing compiler for object-oriented languages," in *Proceedings of ACM Conference on Object Oriented Programming Styles 1996*, 1996.

[28] R. Cytron and D. Shields, "FRIL - a fractal intermediate language," Tech. Rep. wucs-93-51, Computer Science Department, Washington University, July 1993.

[29] M. M. Brandis, *Optimizing Compilers for Structured Programming Languages*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 1995.

[30] Z. Shao, "Typed common intermediate format," in *1997 USENIX Conference on Domain-Specific Languages*, (Santa Barbara, CA), Oct 1997.

[31] C. League, Z. Shao, and V. Trifonov, "Representing Java classes in a typed intermediate language," in *Proc. 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, ACM Press, September 1999.

[32] C. League, V. Trifonov, and Z. Shao, "Type-preserving compilation of featherweight Java," in *Foundations of Object-Oriented Languages (FOOL8)*, (London), January 2001.

[33] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi, "Marmot: An optimizing compiler for Java," Tech. Rep. MSR-TR-99-33, Microsoft Research, June 1999.

[34] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lama, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of CASCON '99*, 1999.

[35] P. Pominville, F. Qian, R. Valle-Rai, L. Hendren, and C. Verbrugge, "A framework for optimizing Java using attributes," in *Proceedings of CASCON 2000*, 2000.

[36] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying Java bytecode for analyses and transformations," Tech. Rep. 1998-4, Sable Research Group, McGill University, 1998.

[37] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, , and J. Whaley, "The Jalapeno virtual machine," *IBM System Journal*, vol. 39, February 2000.

[38] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, and H. Srinivasan, "The Jalapeno dynamic optimizing compiler," in *Proceedings of the ACM 1999 conference on Java Grande*, 1999.

[39] P. Bothner, "A gcc-based Java implementation," in *IEEE Compcon 1997 Proceedings*, pp. 174–178, February 1997.

[40] C.-H. A. Hsieh, J. C. Gyllenhall, and W. mei W. Hwu, "Java bytecode to native code translation: The Caffeine prototype and preliminary results," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.

[41] P. Chang, S. Mahlke, W. Chen, N. Water, and W. Hwu, "Impact: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[42] S. Mathew, E. Dahlman, and S. Gupta, "Compiling Java to SUIF: Incorporating support for object-oriented languages," in *Proceedings of the Second SUIF Compiler Workshop*, 1997.

[43] H. Kienle and U. Holzle, "j2s: A SUIF Java compiler," in *Proceedings of the Second SUIF Compiler Workshop*, 1997.

[44] "Java tree builder." http://www.cs.purdue.edu/jtb/.

[45] "Java Compiler Compiler (JavaCC) - the Java parser generator." http://www.metamata.com/javacc/.

[46] M. Gupta, J.-D. Choi, and M. Hind, "Optimizing Java programs in the presence of exceptions," tech. rep., IBM T. J. Watson Research Center, 2000.

[47] J. Winter, "The data structure visualizer project," September 2000. Work report, University of Toronto.

[48] J. Fingas, "The nodes graphics library," 1999. Work report, University of Toronto.

[49] R. Holt and G. Farmaner, "PBS: The portable bookshelf - introduction." http://swag.uwaterloo.ca/pbs/.

[50] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, "The software bookshelf," *IBM Systems Journal*, vol. 36, pp. 564–593, November 1997.

[51] R. Holt, "An introduction to TA: The tuple-attribute language," February 1997. http://plg.uwaterloo.ca/~holt/papers/ta.html.

[52] I. T. Bowman, "Architecture recovery for object-oriented systems," Master's thesis, University of Waterloo, 1999.

[53] T.-W. D. Chung, "Generation of software landscape using the zJava compiler," 2001. Undergraduate thesis, Department of Electrical and Computer Engineering, University of Toronto.